RADC-TR-88-319
In-House Report
January 1989

SOFTWARE DEFICIENCY ISSUES CONFRONTING THE UTILIZATION OF "NON-VON NEWMANN" ARCHITECTURES

Paul M. Engelhart

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS: N/A
2a. SECURITY CLASSIFICATION AUTHORITY: N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): RADC-TR-88-319	5. MONITORING ORGANIZATION REPORT NUMBER(S): N/A
6a. NAME OF PERFORMING ORGANIZATION: Rome Air Development Center; 6b. OFFICE SYMBOL (If applicable): COEE	7a. NAME OF MONITORING ORGANIZATION: Rome Air Development Center (COEE)
6c. ADDRESS (City, State, and ZIP Code): Griffiss AFB NY 13441-5700	7b. ADDRESS (City, State, and ZIP Code): Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Rome Air Development Center; 8b. OFFICE SYMBOL (If applicable): COEE	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: N/A
8c. ADDRESS (City, State, and ZIP Code): Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.
62702F	5581	18	TK

11. TITLE (Include Security Classification): SOFTWARE DEFICIENCY ISSUES CONFRONTING THE UTILIZATION OF "Non-von Newmann" ARCHITECTURES

12. PERSONAL AUTHOR(S): Paul M. Engelhart

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
In-House	FROM Jan 88 TO Jun 88	January 1989	104

16. SUPPLEMENTARY NOTATION: N/A

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Parallel Processing, Parallel Computing, Interconnection Networks, Software Engineering, Software Life Cycle
12	05		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This in-house technical report summarizes work that was undertaken to survey and conduct an evaluation of the current state-of-the-art in software engineering issues confronting the effective implementation of non-sequential architecture-based computers. Since these architectures do not conform to the model of the von Neumann machine, they are herein referred to as "Non-von Neumann" computers.

Until quite recently, software engineering technology has focused primarily on the use of conventional von Neumann computer architectures. However, it is apparent that utilization of "Non-von Neumann" computer systems is necessary to many of today's, as well as the future's computational requirements. Advances in computer architecture technology, VLSI technology, and interconnection network topologies, to name but a few, have led to a multitude of relatively inexpensive but very powerful high performance processors. The development and utilization of "Non-von Newmann" systems requires a strong knowledge of the

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL: Paul M. Engelhart	22b. TELEPHONE (Include Area Code): (315) 330-4476	22c. OFFICE SYMBOL: RADC (COEE)

DD Form 1473, JUN 86 *Previous editions are obsolete.*

underlying hardware and software structures. However, the largest problem in utilizing "Non-von Newmann" computers is not in the hardware, but in the software where software engineering remains oriented towards conventional von Newmann methodologies.

Since these machines are highly concurrent in nature, a deep-rooted understanding is necessary to determine exactly what software engineering tools and techniques are necessary for their exploitation. This report attempts to provide the necessary insight into many of the software deficiency issues confronting the utilization of "Non-von Newmann" architectures. Many software engineering disciplines need to be addressed, including the choice of a computer language, the necessity of a truly parallel operating system, the availability and use of existing tools, and the need for the development of new tools and techniques, such as the development of an optimizing compiler. Overall, this report is meant to provide a comprehensive insight into the software engineering aspects of these relatively new computer architectures.

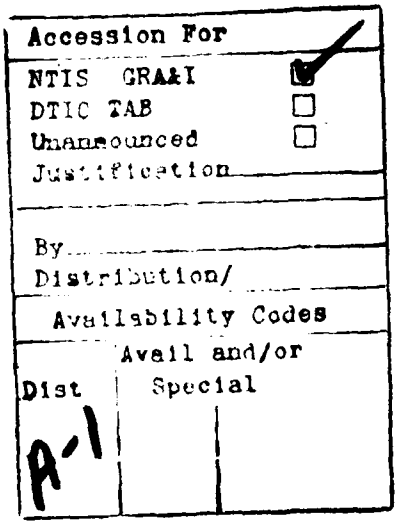

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

This report documents in-house efforts undertaken to understand the problems confronting the utilization of "non-von Neumann" architectures. It is easy to observe that a new era in computer architecture has arrived. This is evidenced by the difficulty to obtain the necessary performance from the conventional von Neumann computer model designed in the 1940's. Hence, there appears to be a rapid transformation from single processor machines, which are typically von Neumann based, to multiple processor machines that emphasize concurrent processing, which have been classified as non-von Neumann. It is increasingly apparent that the utilization of non-von Neumann computer systems is necessary to satisfy many of today's, as well as the future's, computational requirements. The increasing complexity of many applications demand higher computer performance every year. The technological constraints that influenced the von Neumann machine have drastically changed over the years. Advances in VLSI technology have provided a multitude of relatively inexpensive high performance processors. In lieu of utilizing a single computer, new methods have been envisioned to reach solutions through the use of many processors.

Quite simply, non-von Neumann architectures refer to the classification of computer architectures that are not sequential (von Neumann) in nature. Non-von Neumann computers have been designed to meet the processing demands of a wide variety of computationally intensive applications (both numeric and non-numeric in nature) that currently exist in a diverse number of fields, ranging from education to aerospace engineering. However, this report believes that the term non-von Neumann is a bit of a misnomer. This report is concerned with only the truly intended meaning of non-von Neumann architectures that specifically involve some form of concurrency in the processing of instructions and/or data. For lack of a better term, this subset will be referred to as "non-von Neumann" architectures, or "NVN"

architectures. The reason for the distinction is based upon the observation that many current computer systems are by definition non-von Neumann in nature, but do not support concurrent processing. For example, consider MULTICS. MULTICS is basically a large-scale processing system that employs a number of different processors and allows for operations to be done simultaneously which does not adhere to von Neumann programming techniques. Hence, MULTICS is obviously a non-von Neumann computer. However, it cannot be considered a "NVN" computer since it does not support concurrent processing of one application. (A more formal definition of "NVN" computing is provided in section 3.)

To provide a better illustration of what "NVN" computing is all about, consider the following analogy between a computer and a house. The analogy is based upon the notion that the construction of a house is very similar to some of the aspects of concurrent processing. For example, each individual worker in the construction of a house can be thought of as a single processor. In this light, the construction crew as a whole represents a "NVN" computer. Just as several kinds of workers (masons, carpenters, plumbers, electricians, etc) may be employed in the making of a house, it is possible to design a concurrent computer that is made up of several different kinds of processors, each specialized for a particular task. Similarly, just as carpenters must communicate with each other if they are to build walls that line up with one another, so must the processors in a "NVN" computer be able to communicate with one another if they are to work together on a single large problem. The analogy can also be extended to the way work is divided. Imagine that in building a brick house, each bricklayer is assigned a particular stretch of wall. Each worker executes the same operation, but in a different place. Likewise, a computer problem can sometimes be divided in such a way that each processor performs the same operation but on a different set of data. Also consider that when a worker completes his assigned task he can be reassigned another task. A processor can also be reassigned tasks. In this sense the workers as well as the processors are

both reprogrammable. (This analogy will be portrayed throughout the remainder of this report to help describe "NVN" computing.)

"NVN" architectures can consist of tens, hundreds, or even thousands of individual processors that are collectively grouped together. These processors can be comprised of very simple processors (1-bit) to a collection of stand-alone processors, which are very powerful in their own right. It is also possible to mix different processors together in a "NVN" environment. The necessity for obtaining an understanding of "NVN" architectures is based upon the realization that parallel computing is drastically redefining traditional computing. Basically, the "NVN" approach to computing has emerged as a promising architectural alternative to bridging the gap that has appeared due to advances in micro-circuit technology and the increasing costs associated with those advances [35]. The foremost significant strength that the utilization of "NVN" architectures lies in the fact that many applications and problems inherently possess concurrency in their definitions. However, the major obstacle that presently prohibits the effective use of these advanced computer architectures is the identification of the most productive means of applying them. Achieving high performance through the effective utilization of "NVN" architectures does not entirely depend upon using faster and more reliable components, but also on developing major improvements in processing techniques, most of which are software engineering issues.

There are currently numerous non-von Neumann architecture type machines in existence today, with new and advanced architectures being developed at an alarmingly increasing rate. It is apparent that concurrent computing is a very strong candidate for future processing needs, as is evidenced by the use of "NVN" architectures in many areas of today's research as well as in commercial ventures. The development and application of "NVN" systems requires a strong knowledge of the underlying hardware and software structures. A broad knowledge of parallel computing algorithms as well as the optimal

allocation of system resources are also needed. Typically, "NVN" systems entail such architectural classes as parallel processors, array processors, pipeline machines, multiprocessors, vector processors, data flow machines, and systolic array machines. These are discussed in greater detail in section 3.

On the whole, this report is meant to provide insight into the interaction between "non-von Neumann" architectures and the software that they support. As the title of this report suggests, the largest problem that "NVN" architectures currently face is not in the hardware but in the software. Current technology has progressed to the point of providing relatively complex and inexpensive means for building a multitude of advanced computer systems. However, software engineering technology remains oriented towards conventional von Neumann methodologies. This has resulted in a very fragmented software engineering environment. Since these machines are highly concurrent in nature and usually rely upon the specific application to determine their structure and size, a deep-rooted understanding is necessary to determine exactly what software life cycle tools and techniques are necessary for their exploitation. Many software engineering disciplines need to be addressed, including: the choice of high-order languages, the need for new languages, the actual coding in a selected higher-order language, the need or utilization of translators, the necessity for the development of truly parallel operating systems, the availability and use of existing tools, and the need for the development of new tools and techniques such as the development of an optimizing language compiler. Ground rules need to be developed for architecture selection. "NVN" system and software life cycle definitions are needed to support current and future applications. It is also important to understand the ramifications associated with how to best match software engineering technology when the problem domain calls for a hybrid mix, or in other words, when sequential and concurrent processing is utilized in marriage with each other in a single environment. Hopefully this report will provide the necessary insight into many of the

software deficiency issues confronting the utilization of "NVN" architectures.

2. BACKGROUND

2.1. History. It has been observed that since the inception of electronic digital computers in the commercial world, since about 1950, the performance of computing machines has increased at an approximate order of magnitude every five years or so [36].

The underlying rationale for the development of new computer architectures has remained relatively static over the years. Typically, increased computing power and performance, reduced costs, application

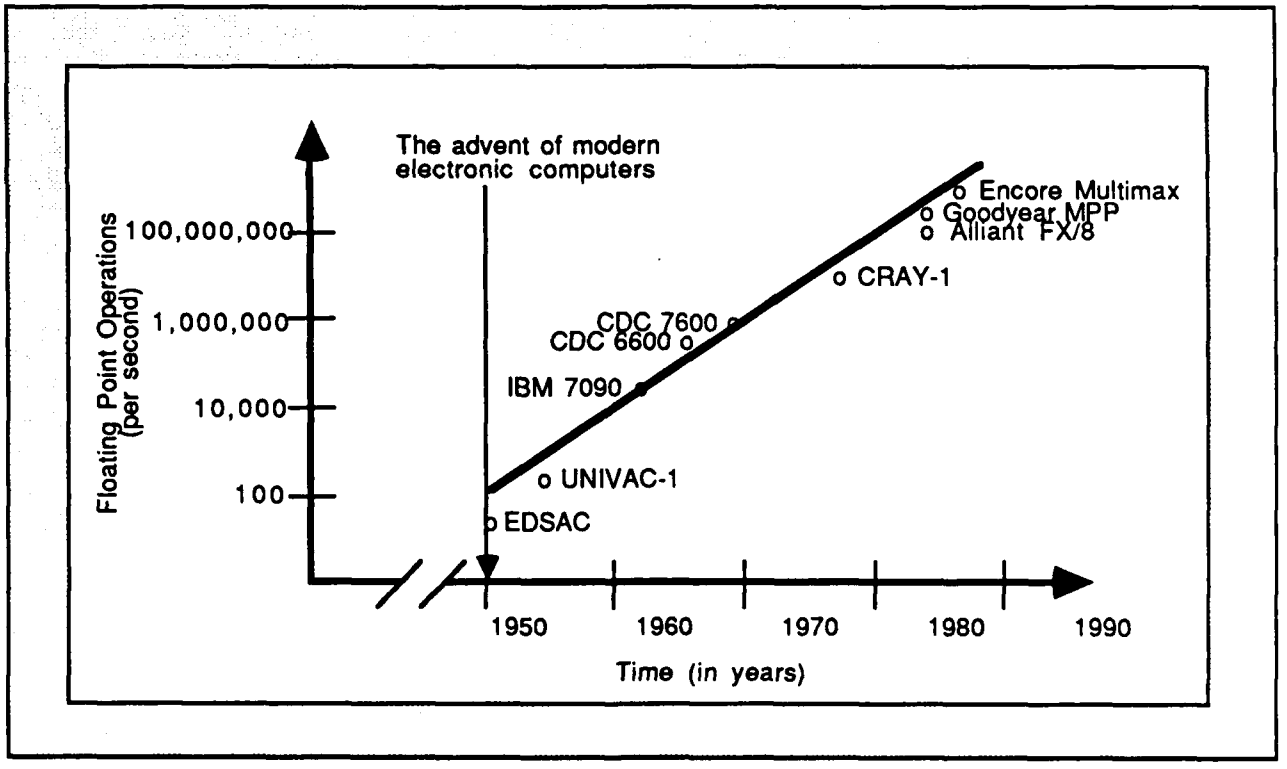

Figure 2-1: The Observed Performance Increase of Modern Computing Machines

support, and advanced programming language support have been the main driving forces [2]. Figure 2-1 is provided to show the rate of advancement in terms of computing power [36].

However, it should be noted that only a percentage of the performance increase can be contributed to the development of new components through the years, such as transistors and VLSI memory chips. Much of the increase is largely due to various advancements that have been made in computer architecture design and development. Obviously many of the emerging advanced high performance computer architectures that are continually being developed utilize new technological advancements, but the underlying computer architectures are the key to the performance increase. As noted previously, however, is that these machines need advanced software techniques to fully realize their potential.

2.2. Computer Generations. The advancements that have been made in computing and computing machinery can be roughly classified into a number of generations (although some machines are extremely difficult to place in one generation over another). Generally, there are a number of driving forces that have been observed which can be used to determine the evolution from one generation to the next, such as the underlying system architecture, electronic component technology, processing mode, and programming languages. The following subsections attempt to trace the progress of computer technology through the years [26], [36], [38]. Many people have tried to classify computer technology advancements into generations, such as the one that this report provides. However, most of the time periods fluctuate between different viewpoints. For example, Hwang defines the period entailing the second generation to be from 1952-1963 while Baer projects that the period from 1958-1964 is more accurate [6], [22]. The computer generations that have been provided in this report are yet another attempt to better pinpoint generation time frames. The information is provided at a relatively high

level of detail so that a familiarity of the rate of technological advancements in the area can be appreciated.

Figure 2-2 portrays the evolution of "modern" computer systems that is derived in this report. It should be noted that adjacent generations may overlap each other since the life span of a generation is intended to include both the technological development of computer architectures as well as the utilization period of the machines within a particular generation.

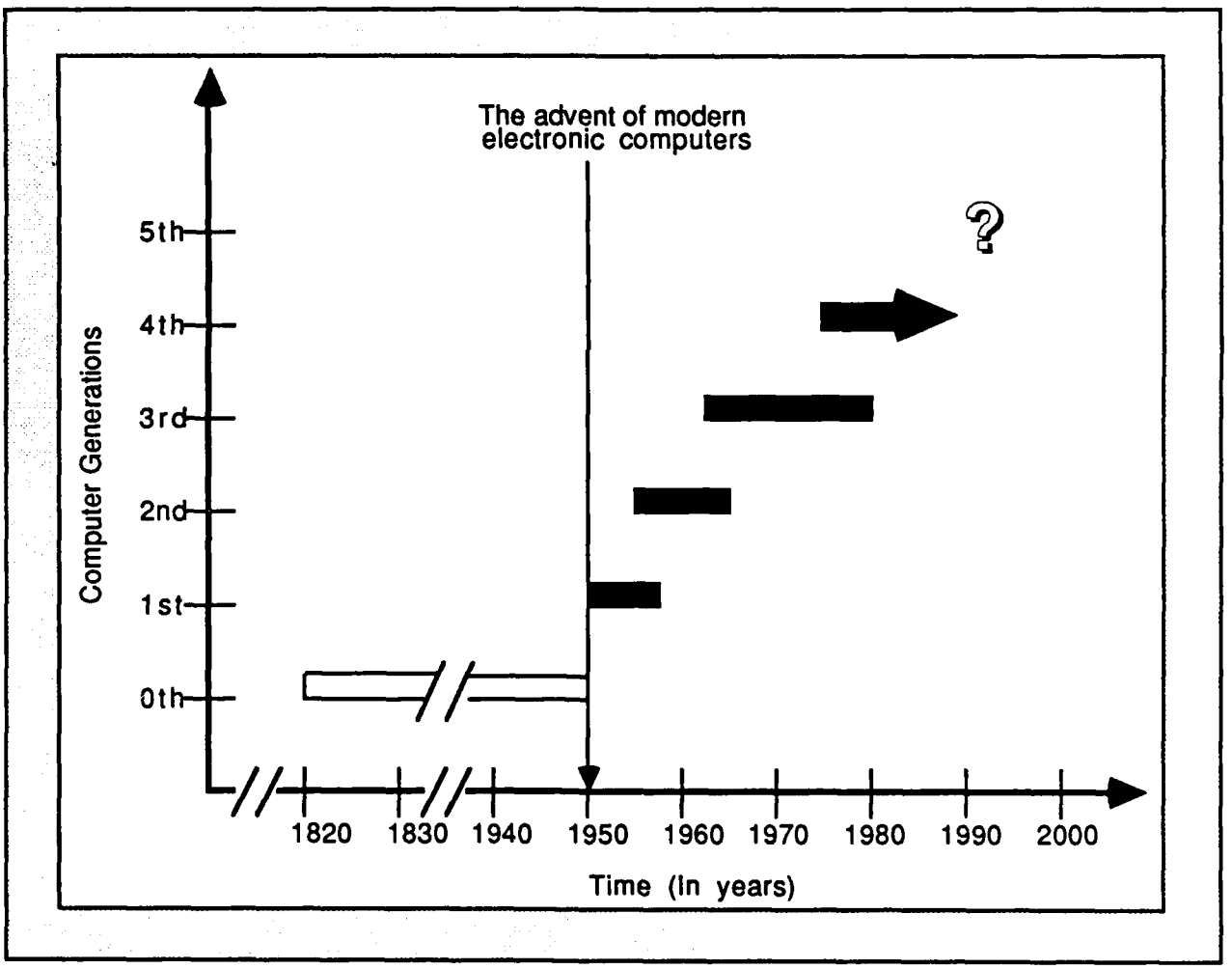

Figure 2-2: The Evolution of Modern Computer Systems

2.2.1. Zeroith Generation. Since the inception of modern electronic computers occurred around 1950, 1950 is considered the beginning of the computer age and will be referred to as the initiation point of the first computer generation. Therefore it is only natural to describe computer system design and development prior to 1950 as being the pre-first-generation of computer architectures, or, as it will be termed here, as belonging to the "zeroith generation." The very first electronic computers that were built were based upon the work performed in the 1930's and 1940's. This work laid the fundamental foundation for the advent of the first modern digital computer. (In fact, these first electronic computers were either the result of government supported research centers or academic efforts.) Many of the ideas and discoveries, although considered as great technical advances in the field, were mainly built upon work done by Charles Babbage nearly 100 years before. Hence, the "Babbage" machine that was designed in the 1820's and 1830's is generally referred to as the computer of the first general purpose computing machine.

The ENIAC, which stands for the Electronic Numerical Integrator And Computer, is considered to be the first electronic computer to actually have been developed. Finished in 1946, the ENIAC was composed of 18,000 vacuum tubes, 1500 relays, and used 150 kilowatts of power. Designed for the purpose of computing ballistic trajectories, it was built in a U-shaped configuration that was over 100 feet long (overall) and 8 1/2 feet in height [22]. The machine was wired specifically for one computational application at a time and any changes or modifications necessitated rewiring the computer. Hence, the set-up time was very long, typically anywhere between one-half hour to one day [26].
Also of significant importance during this time frame was the inception of the concept of having a computer program stored in memory. John von Neumann, who is credited with inventing memory stored computer programs, became enlightened with this concept when working as a consultant to the group that designed the ENIAC. Based on this experience, von Neumann and his colleagues designed the first stored

program computer, called the EDVAC (which stood for the Electronic Discrete Variable Automatic Computer). The development of the EDVAC, however, was severely hindered by the departure of key researchers, including von Neumann. Subsequent to its fabrication, the research that went into the design of the EDVAC gave birth to a number of other machines and ideas. One was the work that von Neumann undertook after leaving the EDVAC project. He and his collaborators designed and built the IAS (Institute for Advanced Sciences) machine which is more commonly referred to as the von Neumann machine [6]. Since this particular machine is the frame of reference for what are now considered serial computers, the definition of non-von Neumann machines or architectures is can easily be derived (provided in section 3).

2.2.2. First Generation. First generation computers can be (arguably) grouped into the time frame encompassing the years from 1950-1957. It was during this generation that the commercial electronic computer industry essentially came into existence. Hence, it is generally accepted as *giving birth to the computer industry.* Computers designed and developed during the first generation still utilized the vacuum tube and electromagnetical relay technology of the 1930's and 1940's to implement logic and memory. Generally, first generation computers were designed for scientific processing applications and utilized machine language programming. The major computer architectural advances that transformed computer technology out of the zeroith generation and into the first generation can be typically associated with the design and development of the UNIVAC (UNIVersal Automatic Computer) machine. The major computer architecture advancement of the UNIVAC was its tape system - magnetic tapes that had the capability to be read forward and backward, which also provided the capability for buffering and error-checking [6]. Many models of the UNIVAC machine were eventually developed and the UNIVAC-I was the first successful commercial computer (developed for the Census Bureau).

2.2.3. Second Generation. Second generation machines roughly entail computer architecture advancements that were observed in the period from 1956 to 1964. The movement from the first generation to the second was primarily due to significant advancements in electronic technology. Generally, first generation machines are characterized by vacuum tubes. Second generation machines now employed transistors. By utilizing transistors, computers could be designed and built cheaper, smaller, and with less heat dissipation problems. Other notable advancements that are associated with second generation computers are their utilization of magnetic core memory, mass storage utilization, the use of printed circuits, the introduction of batch processing, input/output processors, and index registers. Second generation computers were now designed for scientific as well as business applications. Various assembly languages and the first high-order programming languages - FORTRAN, COBOL, ALGOL, PL/1, and even LISP - were also developed during this generation. Although going through a number of changes, FORTRAN and COBOL (which was developed under the direction of the DOD) are still considered to be the standard high-level languages for non-academic purposes. (Moreover, LISP is still widely used and is considered a standard for Artificial Intelligence applications.) The IBM 700/7000 series is a typical example of second generation computers [6], [24].

2.2.4. Third Generation. The passage into the third generation of computers was not a clear-cut transition in terms of technical advances as was evidenced in the transition into the second generation. The third generation is typically identified with the advent of integrated circuits and generalized multiprogrammed operating systems and can generally be associated with the time period from 1962 to 1980. During this generation, the size and cost of computer systems significantly decreased due to two related facts - the widespread use of small-scale integrated (SSI) and medium-scale integrated (MSI) circuits, and the decrease of hardware costs. Also, multi-layered circuit boards were now used. Computers of the third generation could generally support efforts in all application fields. Magnetic core memory that was prevalent in earlier

computers now utilized semiconductor memory technology by implementing solid-state memories [6]. Processor design became easier and more flexible due to microprogramming techniques. Multiprogrammed operating systems, timesharing, and virtual memory developments were also being incorporated [22]. Furthermore, the advances made in compiler technology during this period greatly increased cost effectiveness [26].

The IBM System 360/370 series is a typical example of a third generation machine. The major differences between the IBM 700/7000 series and the IBM System 360/370 series were made in the CPU. However, the differences were severe enough that compatibility between the series was non-existent. Other third generation machines include the PDP-8, PDP-11, and CDC 6600 [36].

2.2.5. Fourth Generation. Generally it can be said that we are currently experiencing the fourth generation of computer technology, which commenced around the mid-1970's. The fourth generation has experienced many new and advanced high performance computer architectures. Many of the machines that utilize these advanced architectures are being developed to exploit new technological breakthroughs in both hardware and software, although at this point in time software development is severely lacking when compared to hardware advancements. Typically, fourth generation computers are characterized by large-scale integration (LSI) and very large-scale integration (VLSI) technology to construct logic and memory components. Most of the operating systems for fourth generation machines are timeshared and utilize virtual memory. Parallelism, pipelining, and multiprocessor techniques are also widely prevalent. High-level languages are being extended to handle both scalar and vector data structures [26], [36]. The fourth generation of computers also witnessed the proliferation of small machines, mostly in the nature of personal computers.

2.2.6. Fifth Generation. Generally, fifth generation architectures are aimed at artificial intelligence and knowledge-based systems. Manipulation of massive amounts of information, inference processing and problem solving, and user-friendly interaction via man-machine interfaces are the key design concerns [38]. Most current computers, whether they be von Neumann or non-von Neumann based, possess relatively inflexible I/O interfaces. For systems to become more user-friendly, interaction with users must be achieved at a higher level through such means as speech, pictures, and natural language. The application of technology to problem application representation and solution is generally considered as the major driving force behind the future move into fifth generation design and development. Allowing voice, pictures, and natural language input to be accomplished in real time requires computing power that is not currently available on standard architectures [26]. Hence, the actual transition into fifth generation machines has not occurred yet.

2.2.7. Summary and Observations. It has become apparent that the future in computer architecture design, development, and utilization has seen a sharp departure from the original von Neumann computer model. Many of the changes that have evolved from one generation to the next have provided the necessary impetus for this departure. Most of the changes have severely impacted the speed, flexibility, and cost of the various components that are utilized in today's computer systems. Obviously, this report bases its premise on fourth generation computers since, generally, "NVN" machines can be classified as fourth generation computers. Software development has steadily progressed throughout the evolution of previous generations but has so far been severely lacking in the utilization of many fourth generation computer systems. Software based methods, techniques, tools, and algorithms are sorely needed to fulfill the promised speed and applications that "NVN" machines offer.

It is interesting to note that despite the advances made in computer systems, and the increasing number of individual components that make up these systems, today's computers are really quite basic in terms of

functionality. For example, consider the world's first computer designer, Charles Babbage, who primarily designed in the 1820's what is considered to be the first general purpose digital computer. Called the Analytical Engine, and although never built, Babbage formulated ideas that have been re-utilized over and over again throughout the history of computer architecture development. Although today's computer systems, and their predecessors before them, are primarily considered as direct descendants of ideas that were devised in the 1930's, these ideas were basically rediscoveries of what Charles Babbage had written about in the 1820's and 1830's. In fact, by the mid-1950's, most of what Babbage had proposed had been implemented. Moreover, Babbage's design methods and ideas for machine organization are still being utilized even in current computer systems.

2.3. Today's Processing Demands. As today's processing needs and requirements continue to become more and more complex, enormous processing power is required to meet their associated computational demands. To be able to solve these requirements in a reasonable time, very powerful computer systems are needed. Much of today's current computing equipment is based on the von Neumann type architecture. Since the von Neumann architecture is characterized by sequential processing techniques, utilizing word-at-a-time processing, it has been found to be very successful performing complex computations on relatively small amounts of data. However, this architecture is extremely inefficient when performing even simple computations on large amounts of data. This is due to the fact that a von Neumann type machine does not utilize its hardware efficiently. It has been shown that the performance of a conventional serial computer of this type is severely limited by the data transfer capability between the processor and memory. (The time required to transfer data is usually limited by the speed in which data and instructions can be moved both in and out of memory). The inability to quickly utilize memory at an appropriate speed has been generally referred to as the von Neumann bottleneck. This bottleneck has been a major driving force behind the advancement of

new and advanced computer architectures which are better suited for current processing demands and techniques.

As mentioned in section 2.2, many efforts have been pursued to increase the processing speed of computers. For a long time, most speed increases were achieved using faster components while maintaining the same basic computer architecture structure (ie, vacuum tubes were replaced by discrete semiconductors - those discrete components were in turn replaced by integrated circuits - core memory was replaced by integrated memory chips). However, today's components seem to be approaching a practical speed limit - the speed of light. (It is known that signals cannot propagate through a semiconductor faster than the speed of light). To help alleviate this constraint, modern high-speed chips are designed to be extremely small in order to reduce the length of interconnections and thus increase speed [40].

Since faster components alone do not satisfy the current demands for high speed computing, the use of other approaches are being exploited to alleviate the performance limiting bottleneck described above. Typically, the methods for computation speedups that are necessitated by these demands involve both hardware and software [28]. Figure 2-3 is provided to show the basic speedup mechanisms and associated problem areas for "NVN" applications.

Subsequently, "NVN" architecture approaches have been developed to provide the necessary processing power for application domains which require concurrent processing of large amounts of data. These approaches utilize the fact that many computations in a program do not depend on each other, and can therefore be executed simultaneously, allowing the computer to process information ten to one-hundred times faster than conventional uniprocessor systems.

One such approach is to overlap execution, which is generally referred to as pipelining. This approach is extensively utilized in many modern

supercomputers (such as the Cray-1 and Cyber 205). Another approach is the use of a collection of computers coupled in some way to do faster processing, which has been referred to as parallel processing. Many ways to combine computers have been proposed and implemented ranging from multiprocessor systems, where a set of identical and/or different processors are interconnected, to array processors, where a large number of processors execute the same instruction on different sets of data.

	Mechanism	Problem Area
I)	Faster and More Reliable Components	Electrical Engineering Material Engineering Physics
II)	Advanced Architectures (Including Interconnection Networks)	System Design Engineering
III)	Programming - Advanced Languages	Computer Science Computer Programming Mathematics
IV)	Optimizing Compilers	Software Engineering Computer Programming
V)	Software Engineering Environments	Systems Engineering Software Engineering
VI)	Computation Analysis and Testing	Computer Science Software Engineering

Figure 2-3: Computation Speedup Mechanisms for "NVN" Computing

A major advantage of a "NVN" architectural based implementation is the time sharing of processing and peripheral units of a computing system among several jobs. The prime impetus for the original development of non-von Neumann systems arose from their potential for high performance and reliability [28]. Today, it appears that performance, cost, and programmer productivity are the primary driving forces (which are influenced by both hardware and software). It is easily recognized that non-von Neumann systems have a multitude of advantages over the von Neumann system. Non-von Neumann systems can promise a much higher availability of resources since they operate as storehouses of resources organized in sequential classes. They also possess a great reserve power which, when applied to a single problem with the appropriate degree of parallelism, can yield a high performance and extremely fast turnaround time. By utilizing two or more central processing units (with all units operational), each processor can be assigned a specific activity within an overall control program. The failure of any one processing unit would degrade, but not immobilize, the system since a supervisor program could re-assign activities and can reconfigure a system so that a deficient processing unit is logically "out" of the system configuration. The failure of a processing unit in a conventional von Neumann system, however, would disable the processor completely. Also, the fact that surplus resources can be applied to other jobs (so that the system is potentially ultra-efficient) produces a very high utilization of available hardware.

Sharing in non-von Neumann processing systems is not, however, limited to only hardware. For example, many systems utilize common use of data sets that are maintained in a system library or file structure. This may represent significant savings in memory storage space as well as in the processing time associated with I/O and internal memory hierarchy transfers. Another advantage of many of today's non-von Neumann computers is their intrinsic modularity. This facet allows for the expansion of the basic system architecture in which the only effect of

expansion on the user is improved performance. Many times the expansion of a non-von Neumann system consists of nothing more than networking identical processor-memory modules. Before, if a user needed additional computing power on a von Neumann based machine, they would typically have to purchase a larger machine or physically reconfigure the current system configuration (ie, add memory, upgrade I/O capability, etc). This is not the case for most non-von Neumann based machines since add-on modular growth without system restructuring is usually possible.

2.3.1. Ramifications. Where does today's software engineering technology stand in relation to "NVN" computer utilization? Until recently, software engineering technology has focused on the use of conventional von Neumann computer architectures and non-von Neumann architectures that do not support concurrent processing. As previously mentioned, current computer systems typically feature increasing computational resource requirements which have become unfeasible or unrealizable using conventional von Neumann techniques. The advent of "NVN" architectures has provided the necessary impetus to offer solutions that can achieve this application requirement. Since "NVN" architectures feature multiple processing elements, they possess a much greater utilization factor than sequential architectures characterized by von Neumann machines. Because of this, a multitude of application areas can be addressed which were once not possible. However, manufacturer's are increasingly concerned that they are unable to commercially sell their products. The problem lies primarily in software. Quite simply, there does not exist enough "real" software, nor software engineering tools, to effectively support these machines.

The field of "NVN" computing is at an extremely critical juncture. If the pace of progress for these machines is to continue, software development holds the key to success. From a system standpoint, such topics as how processors are coupled, how they communicate, how they perform computations, and how an application needs to be broken down,

are very important considerations. The software development and engineering mechanisms to perform these functions are just as critical.

3. BASIC CONCEPTS

3.1. "Non-von Neumann" Computers. What exactly is a "NVN" machine? Quite simply (as alluded to in section 1), the term non-von Neumann computers refers to the class of machines that are not based upon the architecture of a sequential machine. (Also mentioned earlier in this report was that the notion of having a program stored in memory is generally attributed to John von Neumann, hence the term von Neumann computer). This report is concerned with the subset of non-von Neumann architectures that support concurrent processing, which this report denotes as "NVN".

It should be noted that there exist two conditions for supporting "NVN" computing as it is defined in this report. For one, a non-von Neumann architecture is necessary to provide the required hardware vehicle to support concurrent processing. Secondly, the sufficient mechanisms to allow the hardware to permit concurrent processing to occur, such as communication and synchronization methods, must be present. Hence, for "NVN" computing to occur, there must exist both necessary and sufficient conditions. Obviously then, the utilization of a non-von Neumann machine does not in itself provide the means for "NVN" processing. To effectively utilize the hardware, software mechanisms must also be employed. However, it will be beneficial to explain what a conventional von Neumann serial computer is in order to understand what a non-von Neumann machine is not. Once this is accomplished, the subset of "NVN" computers will be easier to understand. Therefore a detailed description follows [6], [9], [43].

Specifically a von Neumann computer exhibits an architecture that consists of the five basic elements (von Neumann referred to them as "main organs") portrayed in Figure 3-1 below.

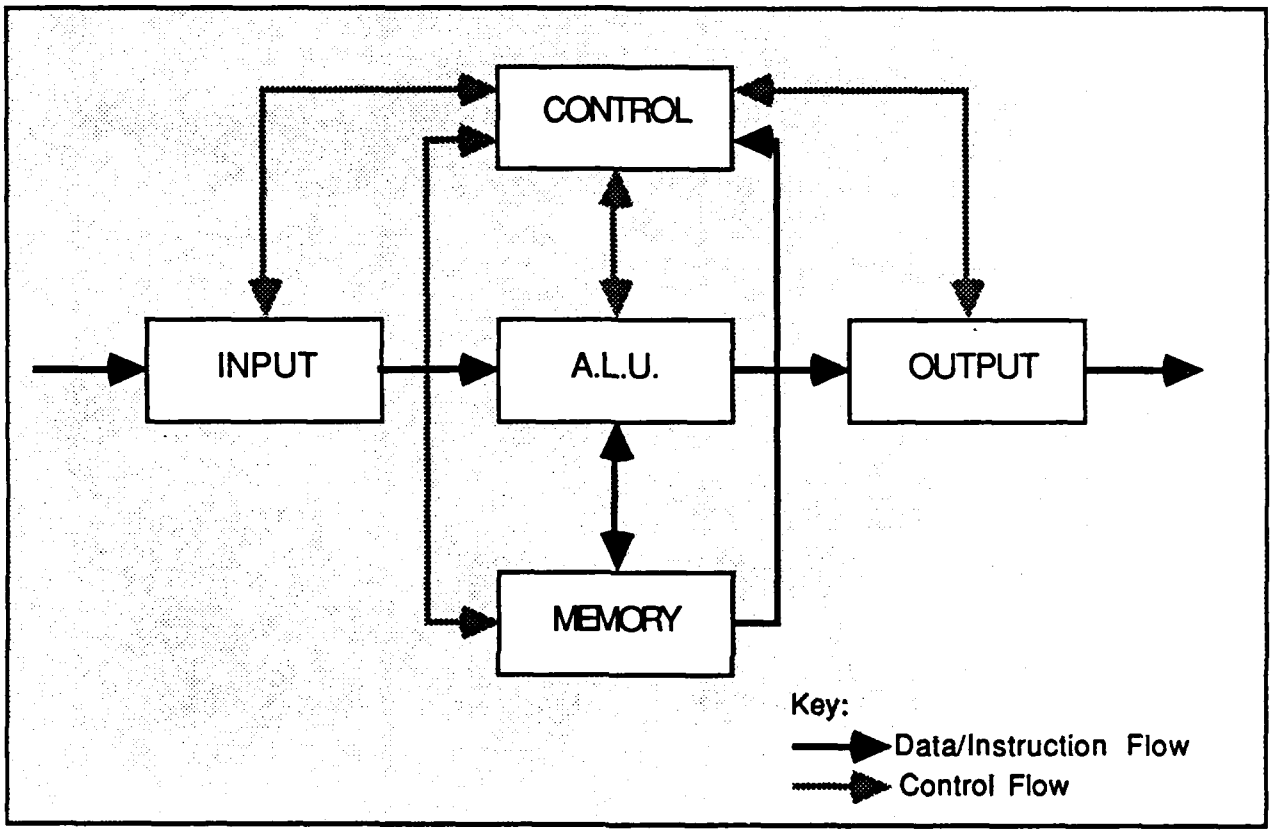

Figure 3-1: The von Neumann Machine Architecture

To put words to the picture, the following dissertation is also provided to help further describe the specific components of a von Neumann machine architecture:

1) Input. The input element transmits data and instructions from the outside world to the memory. This allows a user or operator to communicate with the computer.

2) Memory. The memory element stores the data and instructions that are transmitted via the input. Information used during the computational period performed by the computer are also stored in memory. Conceptually, there are two different forms of memory: the storage of numbers and the storage of instructions (called "orders" by von Neumann). Von Neumann formulated the concept that if the instructions of a computer could be reduced to numerical form, and the computer could distinguish between a number and an instruction, then the memory element could be used to store both numbers and instructions, hence, the genius of the first memory stored program machine.

3) Control. The control element sequences and controls the operation of the computer. Specifically, the control circuitry of the computer interprets the instructions of the program (stored in memory) and then orders the arithmetic logic unit to perform the necessary operations.

4) Arithmetic Logic Unit. The arithmetic logic unit (ALU) performs arithmetic and logical operations on the data that is stored and fetched from memory. The control element tells the arithmetic logic unit which operation it is to perform and also supplies the necessary data for it.

5) Output. The output element transmits the final results and messages to the outside world, which are usually coordinated by the control unit. This event occurs when a computation has concluded or when the computation has progressed to a previously determined point.

Basically then, a non-von Neumann architecture does not adhere to this methodology. To further narrow down this description, "NVN" processing is a computing technique that involves two or more interconnected processors (which can be von Neumann in nature) that concurrently perform different portions of the same application.

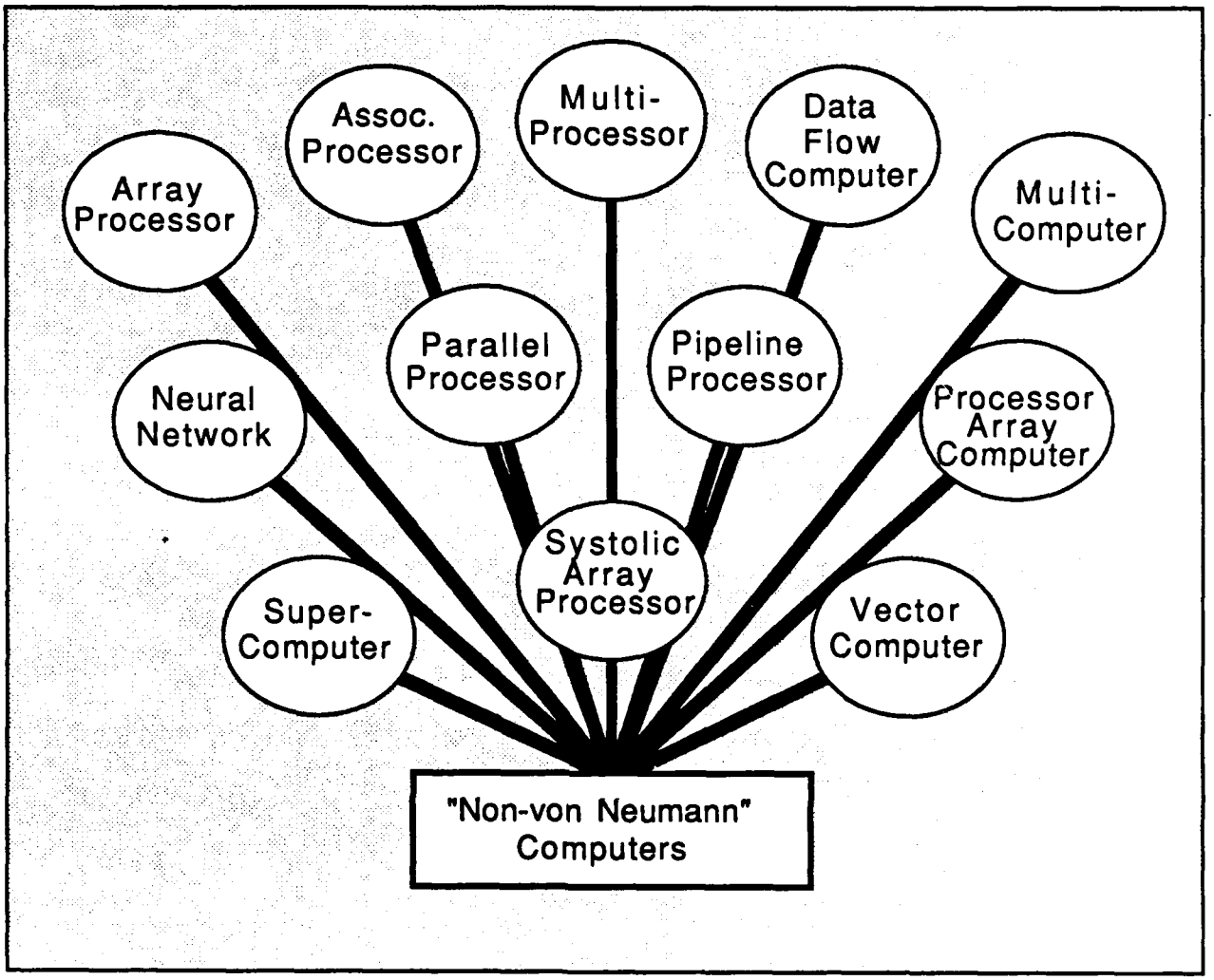

Figure 3-2: "Non-von Neumann" Computers

As seen in Figure 3-2, there exists a wide variety of computers that can be classified as non-von Neumann. Since this report is intended to provide insight into the concurrency issues of non-von Neumann computing, the following subsections briefly describe the types of machines whose architectures are classified as "NVN" [6], [10], [15], [22], [25], [26], [36].

3.1.1. Array Processor. An array processor is nothing more than an attached processor that uses pipelining to manipulate vectors, but without utilizing vector instructions. Array processors are also

referred to as attached processors and are considered a low-cost alternative to a vector computer.

3.1.2. Associative Processor. An associative processor is basically a processor with an associative memory. Instead of having a random access memory, an associative memory permits the entire memory to be simultaneously searched. The Goodyear Aerospace STARAN is an example of an associative processor.

3.1.3. Data Flow Computer. A data flow machine possesses an architecture in which the sequence of instruction execution depends not on a program counter, but on the availability of data. An instruction executes when, and only when, all the operands required are available.

3.1.4. Multicomputer. A multicomputer is a multiple-CPU computer designed for parallel processing, but which does not utilize a shared global memory scheme. Instead, each processor has its own private memory. All communication and synchronization between the processors is accomplished via message passing. The Ametek S/14 and the Intel iPSC are examples of multicomputers.

3.1.5. Multiprocessor. A multiprocessor is a shared memory multiple-CPU computer designed for parallel processing. Unlike multicomputers, multiprocessors are allowed to directly share main memory. The processors in a tightly-coupled multiprocessor set-up employ a central switching mechanism that allows them to utilize shared global memory. The Delencor HEP, Carnegie-Melon's C.mmp, and Encore's Multimax are examples of tightly-coupled multiprocessors. Loosely-coupled multiprocessors, on the other hand, share memory space by combining the local memories of the individual processors. Examples of loosely-coupled multiprocessors are Carnegie-Melon's Cm* and BBN's Butterfly machine.

3.1.6. Neural Networks. Neural network architectures are a relatively new type of computer architecture configuration. Basically, neural networks organize processing element activities in a manner that reflects a simplified model of biological neurons and their behavior. Very few machines based on neural network principles have been built.

3.1.7. Parallel Processor. Quite simply, a parallel processor is a processor that is designed for the purpose of parallel computing. More specifically, it is a processor that utilizes information processing which emphasizes the concurrent manipulation of data elements belonging to one or more processes solving a single problem.

3.1.8. Pipeline Processor. Basically, a pipeline processor is a specific example of an array processor. Specifically, a pipeline processor speeds up array computations via simultaneously processing by breaking a complex function into a series of simpler and faster operations.

3.1.9. Processor Array Computer. A processor array computer is a vector computer that is implemented through the use of multiple processing elements. Each processing element has its own local memory which is acted upon by the control unit of the computer.

3.1.10. Supercomputer. A supercomputer is a general-purpose computer that is capable of solving individual problems at extremely high computational speeds, compared with other computers built at the same point in time. This implies that a supercomputer today may not be classified as a supercomputer tomorrow. For example, a typical personal computer (PC), such as the Apple MacIntosh, would have been considered a supercomputer in the 1950's.

3.1.11. Systolic Array Processor. A systolic array processor refers to a collection of special-purpose, rudimentary processing elements with a fixed interconnection network. By replacing a single processing element

with an array of processing elements, a higher computation throughput can be achieved without increasing memory bandwidth.

3.1.12. Vector Computer. A vector computer is a computer with an instruction set that includes operations on vectors as well as scalars, as opposed to a von Neumann computer that allows only for the manipulation of scalar operations.

3.2. Interconnection Networks. To accommodate the use of "NVN" architectures, particular emphasis must be placed on methods to allow processors to communicate and synchronize with one another. Hence, multiple module interconnection strategies must now be considered to replace conventional strategies. It has been observed that conventional interconnection strategies, such as time-shared/common buses, simple crossbar switches, and multi-port memory schemes are not particularly well suited for systems involving a large number of components [22]. To effectively use "NVN" architectures, various interconnection methodologies to connect individual processors to each other and to memory for communication purposes have been implemented. An interconnection network for a "NVN" machine provides the necessary vehicle, through a connection of switches and links, that allows the aforementioned communication to take place [2], [6], [13], [14], [22], [26], [37]. Figure 3-3 portrays the multitude of interconnection network schemes. The determination of the best network for a particular system must be determined by a trade off between cost and performance characteristics such as component cost, potential growth of the network, fault-tolerence capabilities, and hardware configuration.

3.2.1. Topologies. The easiest way to describe interconnection networks is pictorially. Therefore, the following topologies are provided to depict the different types of interconnection networks [22].

3.2.1.1. Processor-to-Processor Schemes. The following diagrams exhibit the nature of interconnection network topologies between

processors, from elementary examples to the more intricate schemes that are prevalent in many "NVN" architectures. Processor-to-processor schemes are also referred to as static networks and are normally

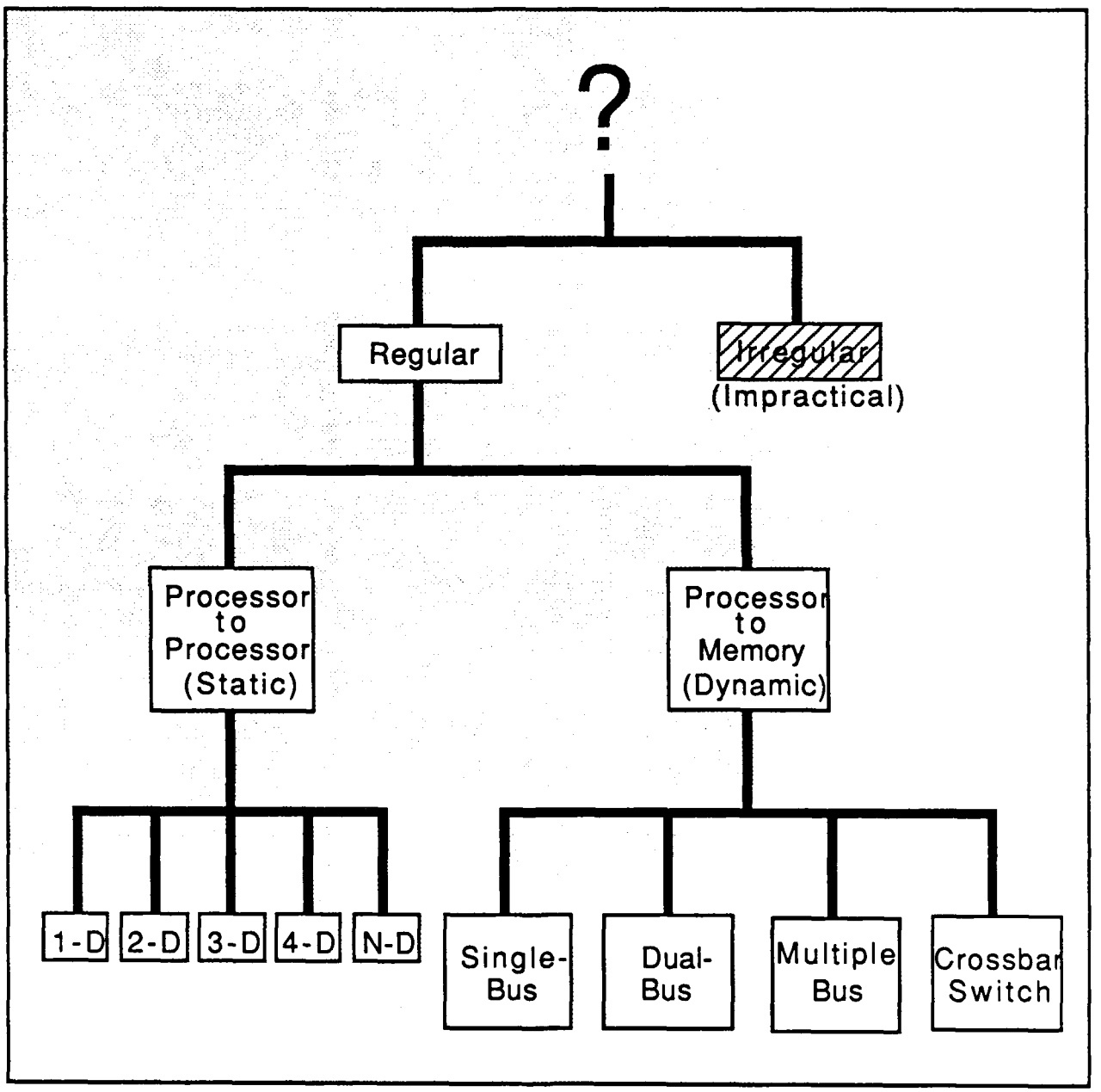

Figure 3-3: Interconnection Network Choices

represented as undirected graphs whose nodes are processors and whose edges are the connections. (Processor nodes are symbolized by small circles; interconnection links are identified by solid lines).

3.2.1.1.1. One-Dimensional Topology: In one-dimensional topologies, the nodes are connected in a line. This network topology is extremely simple but very slow. Figure 3-4 portrays one-dimensional topologies.

In a one-dimensional topology, it should be evident that processors can be arranged in a multitude of different but very unique schemes. The two portrayed in Figure 3-4 are intended to show only two possible configurations. The basic premise is that the processors are all connected in one continuous line. This is very similar to the electrical wiring of a house in a serial manner. For example, when an electrician wires a house, he starts at the source of power (the electrical box where the power enters the home) and runs a wire to a number of outlets and switches. This is done by connecting one outlet/switch at a time. When there is a sufficient load on one run of wire, he simply ends the line and returns to the input to start a new line. Each line is similar to a one-dimensional topology scheme.

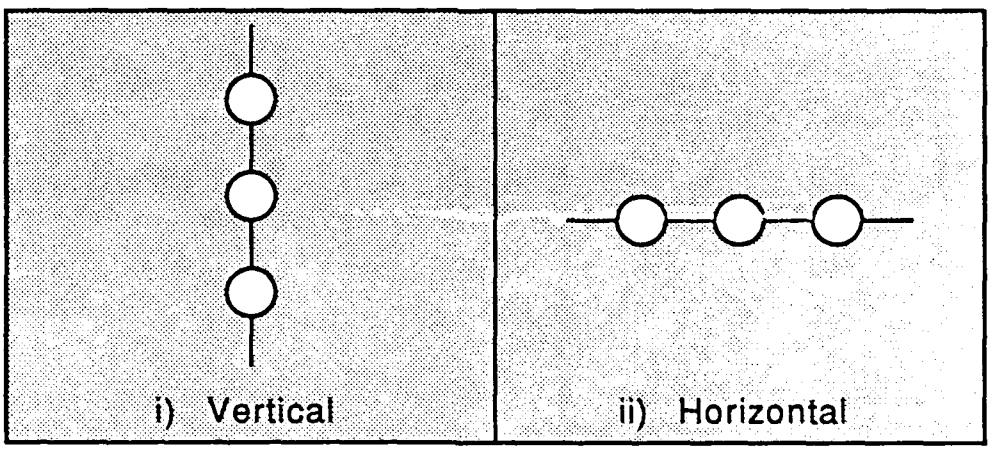

Figure 3-4: One-Dimensional Topologies

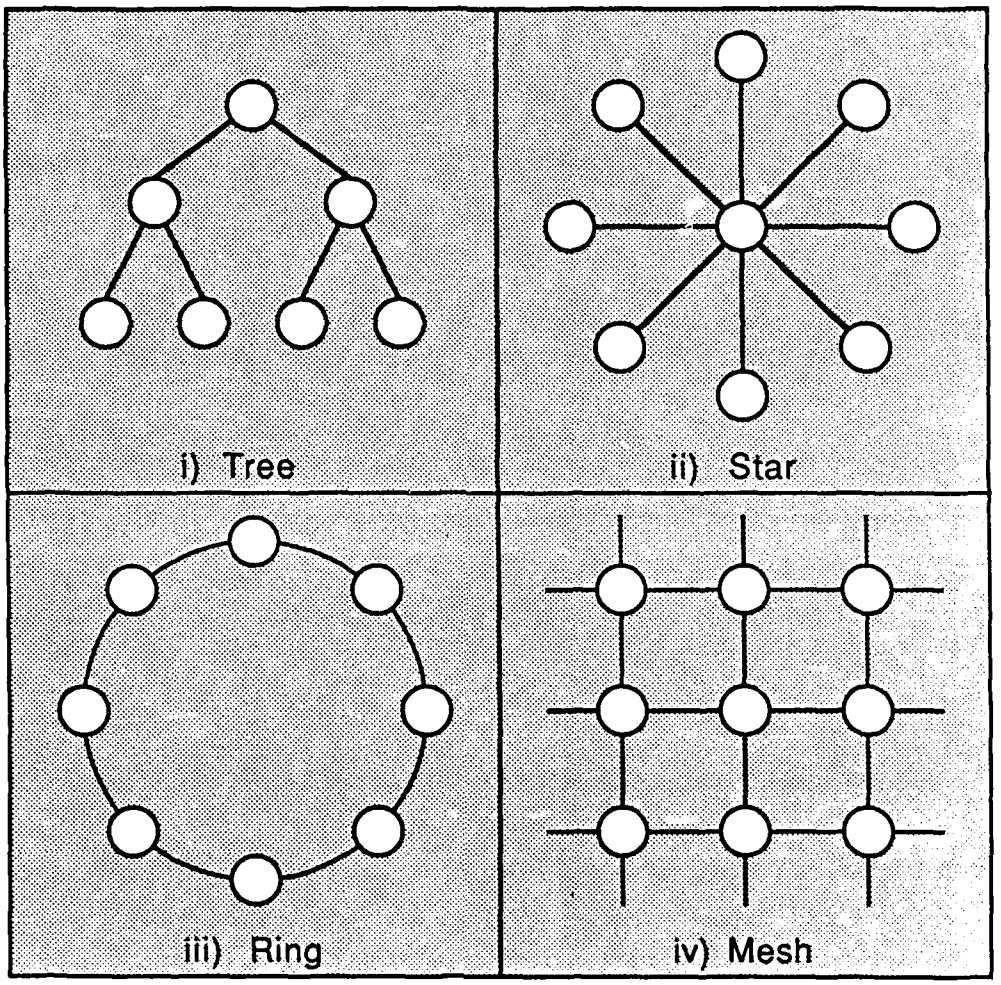

Figure 3-5: Two-Dimensional Topologies

3.2.1.1.2. Two-Dimensional Topology: This class of interconnection networks, displayed in Figure 3-5, represents the most widely used (currently) configurations. The graphs of two-dimensional (as well as one-dimensional) interconnection networks are planar. Due to this they can be easily layed out on an integrated circuit.

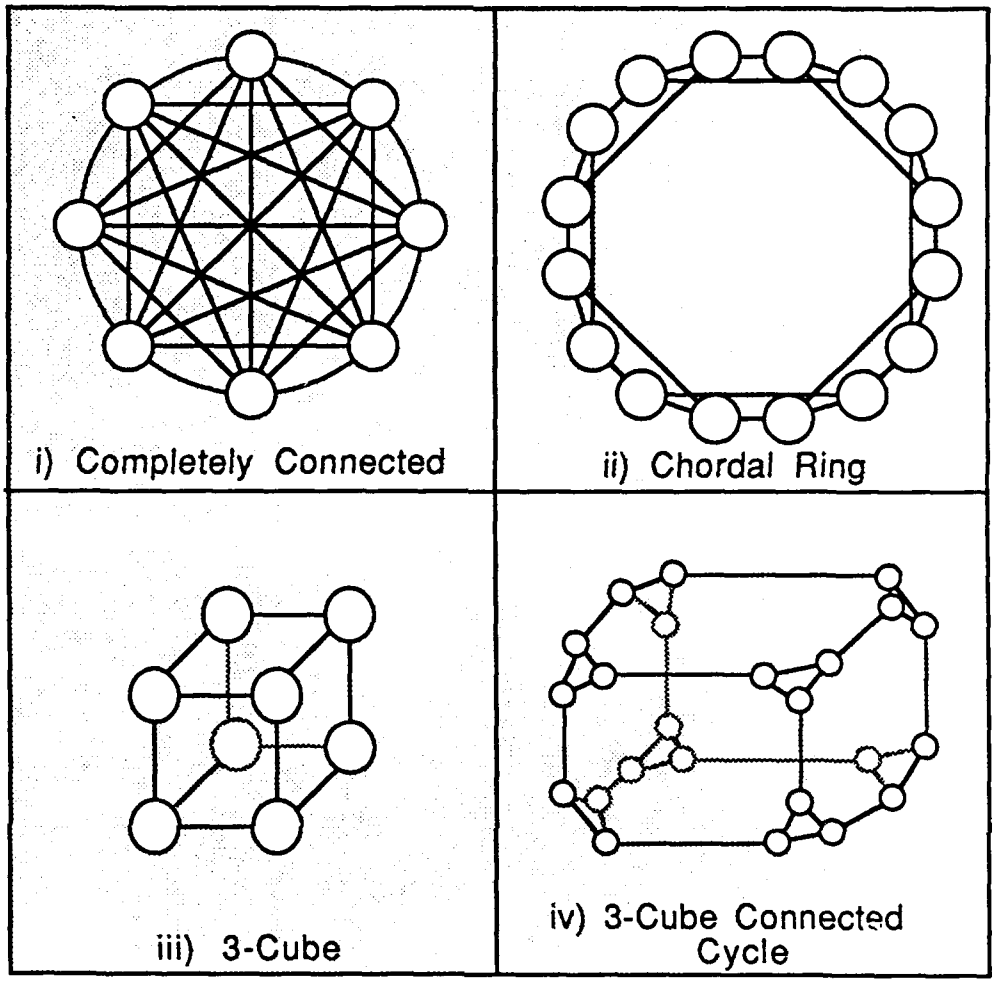

Figure 3-6: Three-Dimensional Topologies

3.2.1.1.3. Three-Dimensional Topology: Three-dimensional topologies are also widely in use today. However, since three-dimensional tolologies are not planar, they are much more difficult to implement. The most prevalent three-dimensional topologies are shown in Figure 3-6.

3.2.1.1.4. Four-Dimensional Topology: These interconnection networks are very difficult to implement, but are becoming more popular. Figure 3-7 shows two variations of four-dimensional topologies.

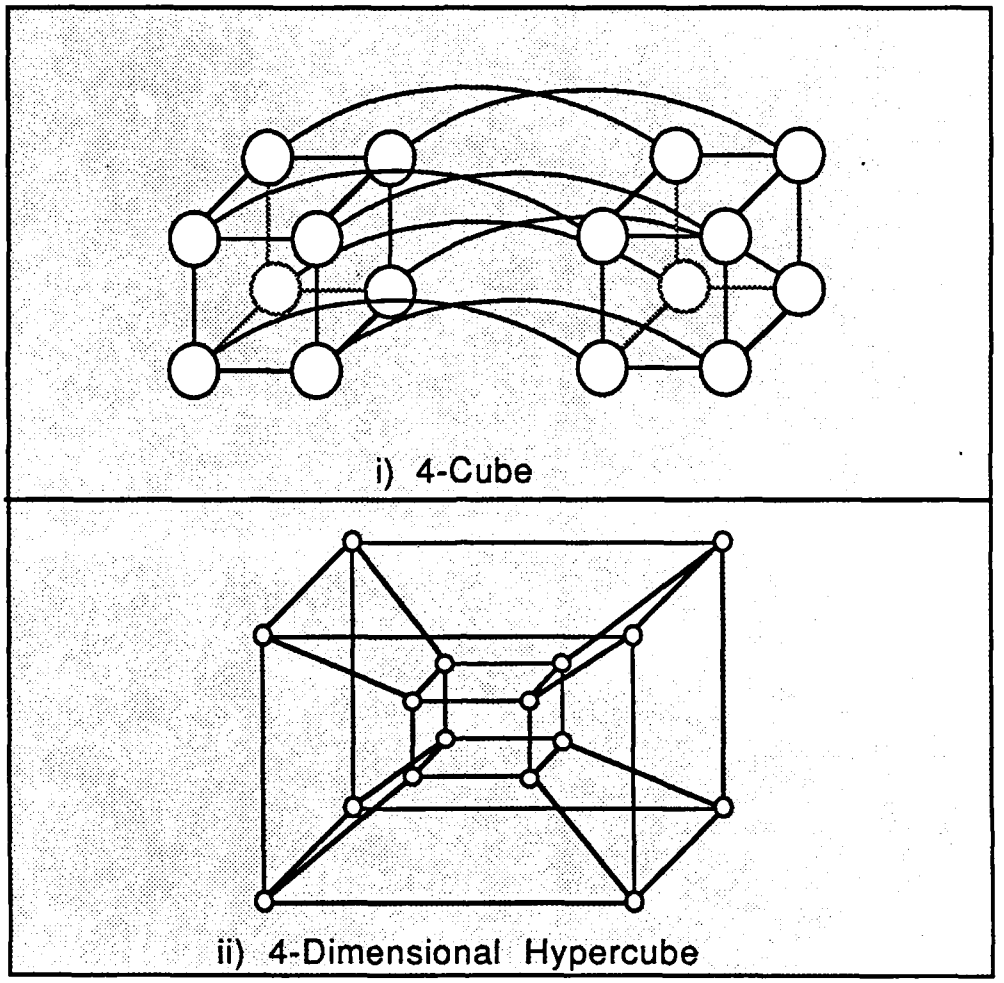

Figure 3-7: Four-Dimensional Topologies

3.2.1.1.5. N-Dimensional Topology. N-dimensional topologies can also be referred to as n-dimension hypercubes. Basically, the topology of an n-dimensional topology (n-dimension hypercube) consisting of x nodes is created recursively from two cubes each consisting of x/2 nodes. For example, using the associated diagrams above, it can be easily shown that a four-dimensional cube is composed of the interconnection of two three-dimensional cubes.

3.2.1.2. Processor-to-Memory Schemes. Processor-to-memory schemes, also known as dynamic networks, are also widely understood. The basic topologies are presented below:

3.2.1.2.1. Single-Bus Organization. Single-bus topologies utilize a shared, bi-directional bus. Of all the interconnection networks, this organization is probably the least complicated. The bus is a common communication path that connect all of the components of a system (processors, memory, printers, etc) but only allows one processor to access the shared memory at a time. Although this configuration is easy to implement, bus contention severely degrades system performance. Figure 3-8 gives a very simplistic representation of a single-bus structure.

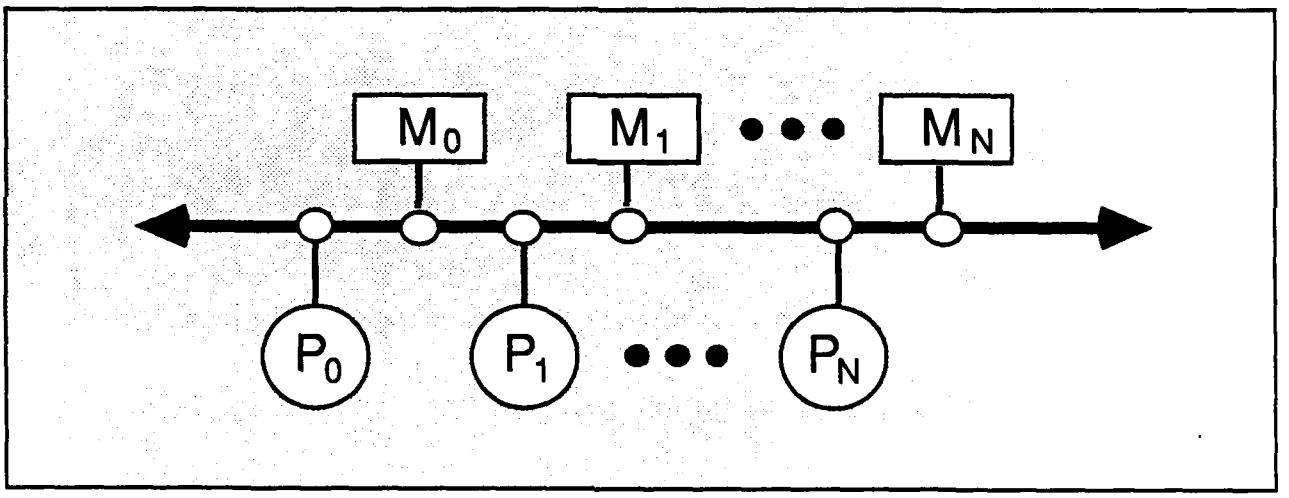

Figure 3-8: Single-Bus Organization

3.2.1.2.2. Dual-Bus Organization. Dual-bus topologies normally employ a two-bus structure, each uni-directional. The extention of the single-bus organization into a dual-bus organization resolves some of the problems inherent to single bus contention problems, although there is still a great deal of bus contention due to the fact that most operations in such a system normally requires the use of both buses. A typical dual-bus organization is shown in Figure 3-9.

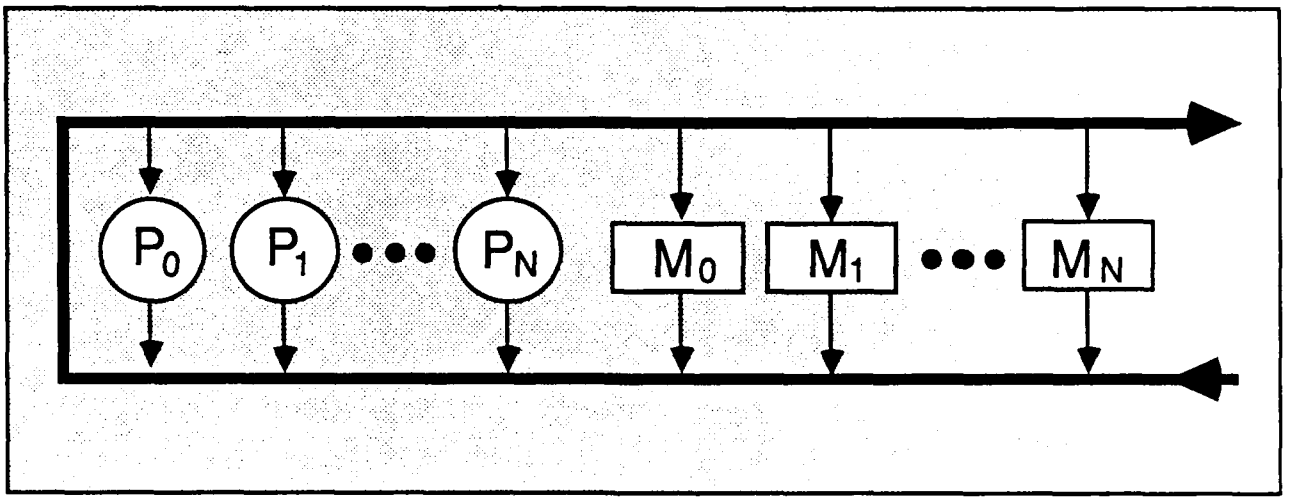

Figure 3-9: Dual-Bus Organization

3.2.1.2.3. Multi-Bus Organization. Multi-bus topologies consist of multiple bi-directional buses. This organization allows for multiple simultaneous bus transfers, alleviating the bus contention problems found in single-bus and dual-bus organizations. Multi-bus organizations are very prevalent in many of today's popular "NVN" type machines. However, there still exists a great deal of overhead time associated with processor-to-memory transfers, memory-to-processor transfers, processor-to-processor transfers, and memory-to-memory transfers.

This organization is shown in Figure 3-10. It should be noted that in this configuration, the number of buses is arbitrary. However, by increasing the number of buses to match the number of processors and/or memory does not guarantee faster throughput. This is due to a number of factors, like bus interference and contention.

This organization looks very similar to the next configuration strategy, the crossbar switch organization. These schemes are not as similar as they appear however.

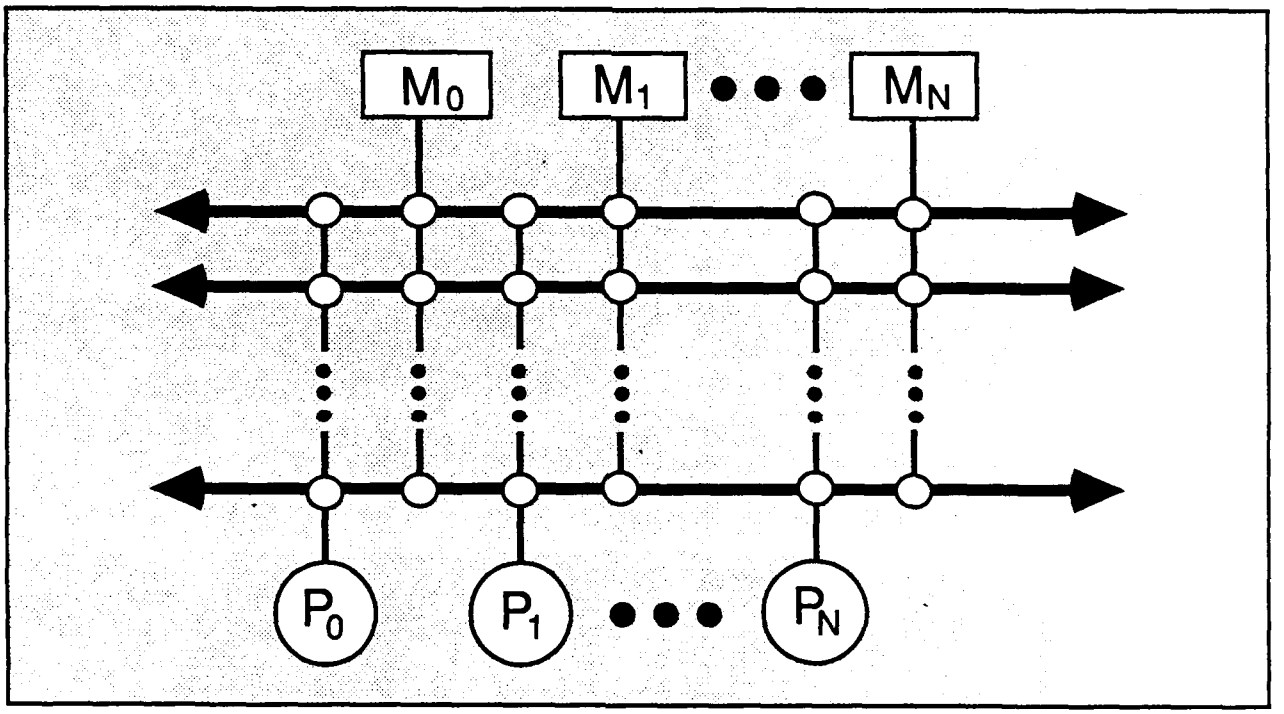

Figure 3-10: Multiple-Bus Organization

3.2.1.2.4. Crossbar Switch Organization. Crossbar switch topologies utilize a grid-like scheme with processors along one axis and memories along another. In this organization, portrayed in Figure 3-11, every processor is logically connected to every other processor. The nodes (portrayed by small circles in Figure 3-11) represent the switches that connect processors-to-processors and processors-to-memory. The switches are capable of connecting cross-wise, hence the name crossbar switch. This type of network is extremely fast.

3.2.1.2.5. Multi-Stage Interconnection Organizations. There exist a multitude of interconnection network topologies that connect processor and memory modules via specialized networks. These topologies are rather intricate to graphically represent (and are therefore not included in the context of this report). Fundamentally, they are all equivalent and differ only in their interconnections between adjacent processors and/or memories. For further detail on multi- stage

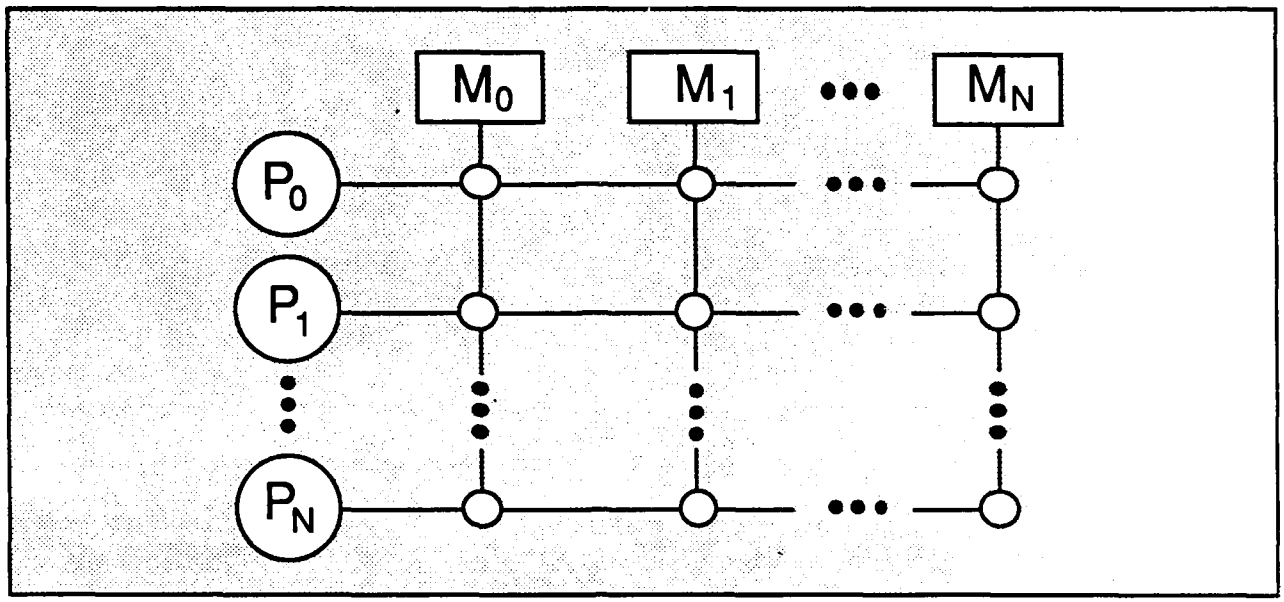

Figure 3-11: Crossbar Switch Organization

interconnection networks, the reader is referred to a number of publications that are referenced at the end of this report. However, to provide a well-balanced cross section of the types of networks that are classified as multi-stage interconnection topologies, consider that the following networks are classified as such: the butterfly network, the omega network, the SW-Banyon network, the baseline network, the flip network, the indirect binary n-cube network, the multi-stage shuffle-exchange network, the generalized cube network, and the extra-stage cube network to name but a few.

3.3. Classification Schemes. A number of classification schemes have been developed that attempt to classify computer architectures [5], [19], [20], [27], [36], [40]. However, the emergence of "NVN" architectures has placed serious problems on many of these schemes. For example, some architectures can be placed in more than one class within a particular classification scheme. It appears that no single classification scheme is sufficient to properly classify "NVN"

architectures. The examination of different system attributes has been the primary driving force behind the classification of computer organizations. Basically, these system attributes have been:

1) the type and number of distinct elements (processors, memory, I/O devices, etc) in a particular computer system,

2) the specific topology of the interconnections (interconnection network) in the system,

3) the specific nature of the processor-to-memory interconnection scheme (local, global), and

4) the intended purpose or application that the system is intended for.

It appears that the development of a "hybrid" classification scheme may be more beneficial. The following paragraphs briefly present the more popular classification schemes for computer architectures.

3.3.1. Flynn's Taxonomy. One of the early (and most widely used) fundamental classification schemes of computer organizations was developed by Michael Flynn in 1966. Probably the most recognized classification scheme, Flynn's taxonomy classifies computer organizations into four classes according to the uniqueness or multiplicity of instruction streams and data streams. The instruction stream refers to the sequence of instructions as performed by the machine, which can be either one sequence (single instruction stream) or many sequences (multiple instruction stream). The data stream refers to the sequence of data that is manipulated by the instruction stream, which also can be singular or multiple (single data stream or multiple data stream). By observing all possibilities for instruction and data streams to be configured, Flynn devised four classification classes which are presented in the following subsections.

3.3.1.1. SISD. A Single Instruction stream, Single Data stream computer, abbreviated SISD, usually consists of a single processor connected to a single memory. A SISD computer fetches its instructions consecutively and then fetches one data item at a time. SISD systems typically entail sequential and uniprocessor computers and are considered the serial (or classic) von Neumann machine. Variations of SISD machines have been implemented that utilize advanced techniques, such as pipelining. However, although instructions may be pipelined, only one instruction can be decoded per unit time. Also, SISD systems can have multiple functional units under the control of a single control unit. Many standard computers fall into this class, such as the DEC VAX 11/780 and IBM 360/370 series.

3.3.1.2. SIMD. SIMD refers to systems that employ a Single Instruction stream, Multiple Data stream configuration and typically consist of a single control unit, multiple processors, multiple memory modules, and an interconnection network. In SIMD architectures, the control unit fetches and decodes an instruction. The instruction is then executed either in the control unit itself, or is broadcast to any of a number of processing elements. These processing elements operate synchronously where all processing elements execute the same instruction at the same time, but with their own data. Processor arrays (pipeline processors, vector processors, array processors, and associative processors) fall under this category. Specifically, the Illiac IV, the STARAN, and the MPP computers are examples of SIMD machines.

3.3.1.3. MISD. A Multiple Instruction stream, Single Data stream (MISD) machine is a computer in which the same data item is operated on simultaneously by several instructions. This mode of operation is realistically very impractical and only a few processors of this type have ever been built. Moreover, no computer systems presently exist that can be categorized uniquely as MISD machines.

3.3.1.4. MIMD. In MIMD (Multiple Instruction stream, Multiple Data stream) systems, all processing elements execute asynchronously on their own data, but share access to a common memory. Hence, each processor follows its own instructions (multiple instruction stream) and also fetches its own data on which it operates (multiple data streams). The coupling of processor units and memories is one feature to differentiate between MIMD designs. Another is the homogeneity of the processing units. Many computer systems fall under this category, including the BBN Butterfly, the Cm*, and the Cosmic Cube.

3.3.1.5. Observations. Flynn's taxonomy naturally encompasses any computer architecture that can be described as executing instructions which operate on data. Although this taxonomy is still in use today, many of the current advanced computer architectures that have emerged have presented problems. This classification scheme is too ambiguous to permit an ironclad labeling of many "NVN" architectures. Depending on one's definition of instruction and data stream, some architectures can be placed in more than one class. For example, the Cray-1 computer has been classified in literature in three different classes. Handler and Thurber classify it as a MISD machine, Hockney and Jesshope classify it as a SIMD machine, and Hwang and Briggs classify it as a SISD machine.

3.3.2. Handler's Taxonomy. Wolfgang Handler's classification scheme, developed in 1977, prescribes a notation for expressing the parallelism and pipelining that occurs at three levels of a computer system - at the processor control unit level, the arithmetic logic unit level, and at the bit-level circuit level. A system is represented by three pairs of integers: C=(K,D,W) where K is the number of control units, D is the number of arithmetic logic units controlled by each control unit, and W is number of bits in an arithmetic logic unit. (For example, the IBM System/370 is denoted as (1,1,32) and the Illiac IV as (1,64,64)). However, this classification scheme has severe limitations in terms of identifying

important features of a computer system, such as instruction and data flow, and memory configuration (ie, shared vs distributed).

3.3.3. Feng's Taxonomy. Developed in 1972, Tse-yun Feng's classification scheme is based upon serial processing versus parallel processing. It uses the degree of parallelism according to the word length of a machine to classify various computer architectures. This taxonomy is not very useful when considering "NVN" systems. The multiplicity of instruction and data streams, the type of memory configuration, and the number of processors are some of the important features not recognizable by this scheme.

3.3.4. XPXM/C Taxonomy. This taxonomy is a relatively new classification scheme that is based upon the classification of the processor coupling techniques used within microprocessors. Basically, XPXM/C refers to single-or-multiple (denoted X) Port, single-or-multiple (also denoted X) Memory/Channel. The different coupling techniques are used to determine the number of simultaneous transactions that a multiprocessor system can support, and provides for a relatively good basis for comparison of multiprocessors. Used in conjunction with other existing classification schemes that describe the topological and functional characteristics of complete multiprocessor systems, the XPXM/C taxonomy helps to emphasize the constraints on parallel system operation. However, this classification is not intended for "NVN" architectures outside the multiprocessor realm.

3.3.5. Other Taxonomies. There are a number of additional taxonomies in existence that help to classify many of the aspects of "NVN" computing, but which fail to provide a full-scaled comprehensive classification scheme. These include such taxonomies as Haynes (1982), Schwartz (1983), and Mak (1986). There are also a few subtaxonomies that exist that are useful to classify specific types of operations such as Treleaven (1982) and Srini (1986) which are useful for data flow architectures. Hockney and Jesshope (1981) present a taxonomy that is

good for processor arrays. Yau and Feng (1977) present a taxonomy that provides insight into associative processors.

4. SOFTWARE FOR NON-VON NEUMANN ARCHITECTURES

4.1. General Considerations. Once a decision has been made to utilize "NVN" architectures, a number of very important considerations are placed on the development of software for these architectures. The first consideration, to map a particular problem or application for a "NVN" architecture implementation, the computer programmer must partition the problem into a number of concurrent units and then decide upon the preferred communication/synchronization mechanism. This is true no matter what the problem/application or what "NVN" machine is chosen. The key issues and technology areas that are associated with software for "NVN" architectures can be broken down into a number of distinct areas, namely: problem/application decomposition, communication/synchronization, parallel algorithms, languages, compilers, and operating systems. These are discussed in greater detail further in this section and depicted in Figure 4-1.

It has become increasingly apparent that the development of software for many applications is by far the most complex and time consuming. Hardware is being designed and developed at a much faster rate than software. It is also a fact that the hardware no longer encompasses the greatest cost of a computer system. Software development now typically costs several times more than the development of hardware. Also, software usually out-lives the life of a normal hardware configuration. This report supports the notion that the most successful approach for developing software for "NVN" architectures may employ a "software first" methodology. This approach basically employs a philosophy that focuses on software development first, and then focuses on the development of the appropriate hardware that will effectively support the execution of the software.

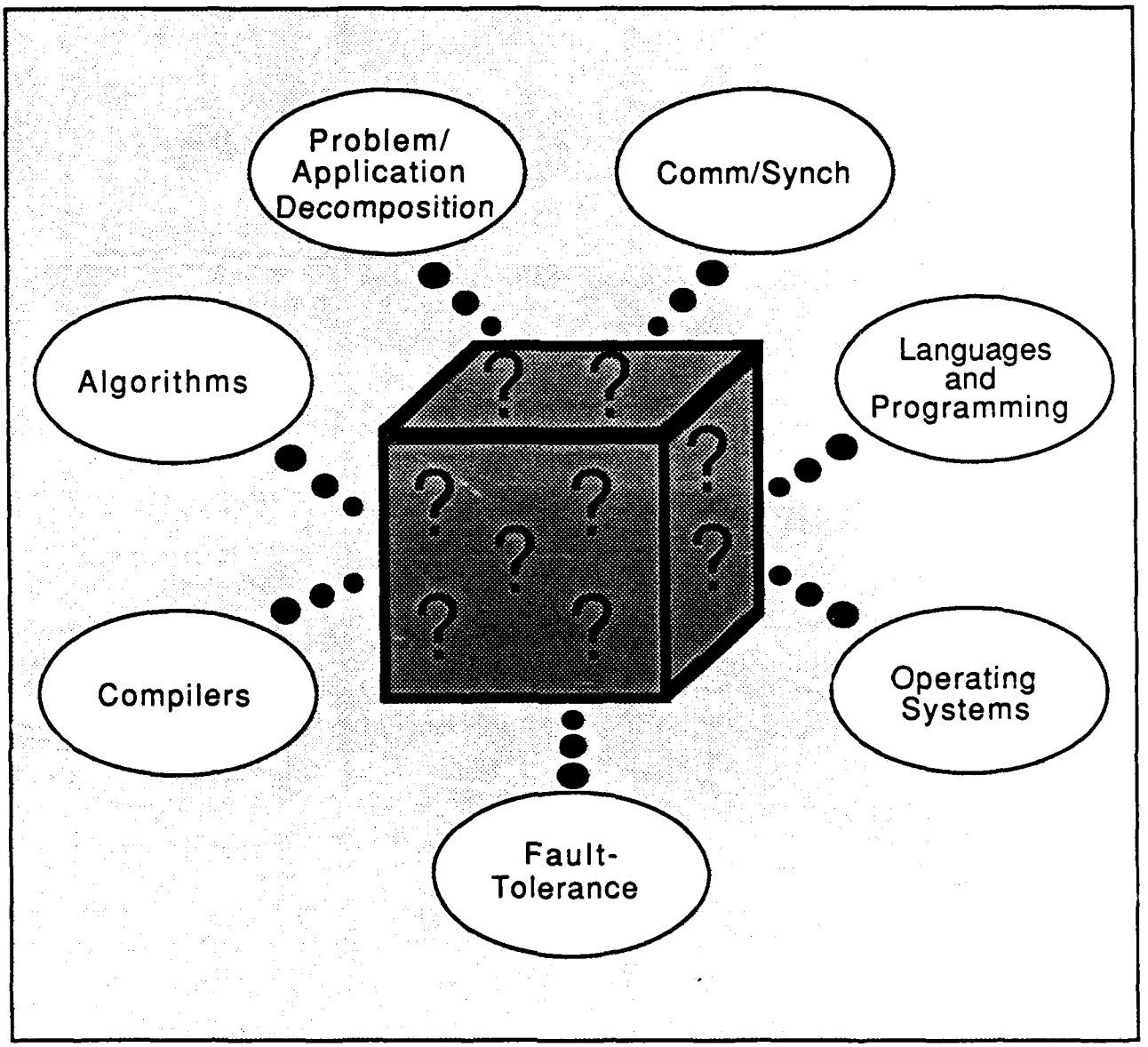

Figure 4-1: Software for "Non-von Neumann" Architectures

4.1.1. Typical Computer System Utilization. The first observation that should be made is how a typical "NVN" computer system is devised, and how does it compare to a von Neumann system configuration. Figure 4-2 portrays a very vivid overview of how computer systems are used.

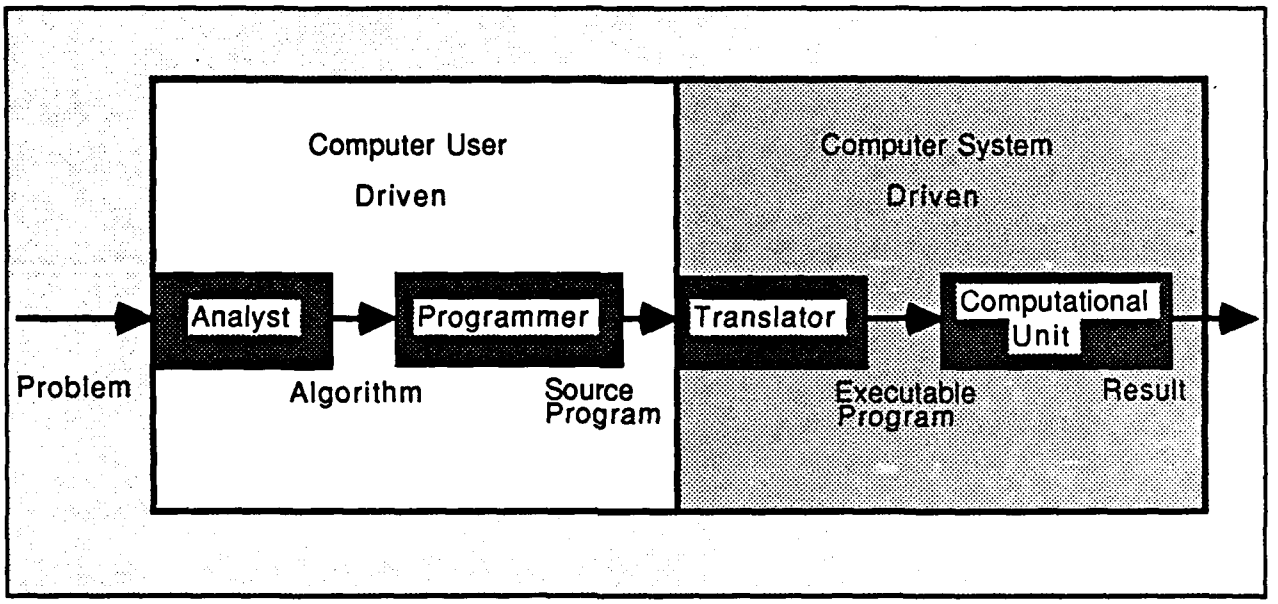

Figure 4-2: Typical Computer System Utilization
Sequence of Events

Generally, it can be observed that this configuration holds true for any computer system, whether it be von Neumann or non-von Neumann in design. Once a problem has been determined, a computer analyst examines the problem using some kind of known methodology. The analyst then comes up with the necessary algorithm to solve the problem. After the algorithm has been developed, the algorithm is programmed via some programming language that supports the algorithm and associated data structures. Next, the program is translated and the executable code that is produced is run on the computer system together with some data and produces an output result.

4.1.1.1. Possible Choices. It should also be noted that at each stage in the utilization of a computer system (as portrayed in Figure 4-2), solutions can be accomplished differently. Obviously, some are better than others. For example, in the design of an algorithm, the desired speed and memory space requirements are important considerations. In programming there exists a number of design decisions. As an example, a

programmer has a multitude of programming languages to chose from. He/she may choose the one that is most familiar or comfortable for their use, or may choose a language that inherently displays the capabilities to better solve the problem. Also, every programmer has their own individual programming style [20]. Clearly, there are a number of possible computer programs that could result from a single algorithm. Many of these issues are discussed in greater detail in the following subsections.

4.2. Problem/Application Decomposition. The purpose of designing any kind of computer system is to serve some audience of users, whether it be for general-purpose use or directed towards a specific application. At this point in time, it appears that "NVN" computers are predominately the latter. In fact, the design factor considerations of many of today's "NVN" architectures were based on a specific users desires and needs. It is therefore very important to initially segment a problem or application into units that will execute in a parallel nature, with no emphasis being placed on a particular architecture or machine [29]. At this point, the analyst is only concerned with devising a very high-level decomposition of the problem. The selection of a particular "NVN" machine will be emphasized later.

4.3. Communication/Synchronization. Next, the analyst must decide upon the mode of processor communication and synchronization between each other. Again, this is only done at a very high-level. At this time no importance is placed upon a particular architecture. However, decisions made for communication/synchronization may play a very important part in the eventual decision to pick a particular class of machines [15], [23].

4.4. Designing Parallel Algorithms. Designing parallel algorithms can be a very complex and time-consuming undertaking. The use of existing algorithms can also be very difficult. Algorithms that were originally designed for a von Neumann based (sequential) machine which are now redesigned for a "NVN" architecture may not even resemble each other,

even though they are solving the exact same problem [24], [36]. There has been very little effort at establishing general techniques for the formulation of parallel algorithms [8]. Moreover, many different algorithms can be utilized to solve a single problem. For example, consider the single problem of sorting a set of numbers. There exists a number of unique algorithms to solve this problem, ranging from a simple bubble sort to a more complex perfect shuffle algorithm. The point here is that the same problem can be solved via different approaches. Basically there are three methods that can be utilized to design a parallel algorithm:

1) Utilize an existing sequential algoritnm and exploit any inherent parallelism that can be found in order to formulate a "parallel" version of the algorithm for a particular problem,

2) Utilize an existing parallel algorithm that solves a similar problem and adapt the algorithm to solve the problem at hand, or

3) Design a new parallel algorithm.

It must be understood that there is not an easy and straightforward approach to selecting the "right" method for parallel algorithm development for a particular application. Each method has its own advantages and disadvantages. For example, by utilizing an existing sequential algorithm, it may be possible to profit from work that has already been done by someone else. However, if a complete understanding of the underlying problems that can arise from transforming a sequential algorithm are not examined, many difficulties may occur. Some sequential algorithms possess no obvious parallelism, while others display parallelism that practically jumps out. Also to be taken into consideration is speedup. Even if a particular sequential algorithm can quite easily be transformed into a parallel algorithm, it may exhibit very inadequate speedup and not provide any performance increases over its

sequential counterpart. Therefore it may be wise to design a new parallel algorithm.

4.4.1. Design Considerations. There are a number of considerations that a designer should take into account when designing a parallel algorithm, whether it be an adaption of an existing algorithm or a new design. Again, the process of designing a parallel algorithm can be very difficult [31]. Hopefully, a careful observation of the problems that directly affect the design process of a parallel algorithm will provide sufficient insight into the difficulties of developing a parallel algorithm. Basically, there are five key concerns that a designer must consider:

4.4.1.1. Parallel Thinking. Probably the hardest "process" to initially comprehend is parallelism itself. We live and think in a very logical and sequential world. To suddenly stop and try to logically think in a concurrent way is initially very difficult. It is easy to observe that the vast majority of people appear to be better suited to focus their attention on one activity at a time. Thinking of more than one activity, or thinking in parallel, is very difficult. For example, consider an exercise of trying to read two books at once. First you read one line from the first book, then you read one line from the second, then the second from the first, then the second from the second, and so on. If you continue to follow this alternating pattern for the duration of both books, it is obvious to see that a number of deficiencies may appear (such as reading comprehension). Obviously, thinking in parallel can create many problems.

Once accustomed to focusing on the key concepts of parallelism, a programmer will be able to design much more efficient parallel algorithms. However, learning to program in parallel is very different than learning to program sequentially. As an example, consider the construction of a house analogy. Suppose two carpenters are applying plywood to the frame of a house. Carpenter one measures the size of plywood needed and then informs carpenter two of that measurement so

he can cut it correctly. Carpenter two then gives the cut plywood sheet back to carpenter one so he can nail it up. By thinking beforehand about the task to be performed, a great deal of speedup can be accomplished. Consider that while carpenter two is cutting a sheet of plywood, carpenter one is measuring the size needed for the next sheet. Therefore, when carpenter two delivers a sheet to carpenter one for nailing, carpenter one can inform carpenter two the size needed for the next sheet. This is much more efficient than if the sequence of measuring, cutting, nailing, measuring, cutting, etc, was performed. This is also true for a computer programmer. An adaption to thinking in parallel beforehand will allow for designing more efficient parallel algorithms. This leads to insight into parallelism.

4.4.1.2. Insight Into Parallelism. Insight into parallelism is a very important consideration when designing efficient parallel algorithms. Being able to spot where, when, and how to utilize parallelism is very important. If parallelism is carefully and skillfully applied then a more fruitful algorithm will be produced. For example, to solve a problem via a parallel algorithm approach, some prior or acquired knowledge/insight into choosing a method for the design is very desirable. As mentioned previously, the utilization of a good sequential algorithm or parallel algorithm that solves a similar problem can be a very good starting point. If it is determined that the existing algorithm is not particularly parallelizable, then a knowledgeable understanding of the particular problem must be utilized in order to break up the problem into reasonable "chunks" that may be able to be parallelized for designing a new algorithm [20].

Overall, if an algorithm consists primarily of sequential steps, the use of a "NVN" approach is not feasible since not much speedup can be obtained. However, if an algorithm can be decomposed into "chunks" that can be executed in parallel, then "NVN" computing is very desirable. Obviously, the process of algorithm decomposition is very critical to the success of possible parallel processing attempts [45].

Decomposition primarily consists of two issues: partitioning and assignment. Partitioning is concerned with the splitting of an algorithm into procedures, modules, and processes (ie, chunks). Partitioning is considered the key to efficiently extracting the parallelism of an algorithm. Assignment, on the other hand, is concerned with allocating these chunks to available processors. For example, when a house is being built a number of activities can take place concurrently. Roofers, electricians, masons, etc, can all be performing their associated tasks at the same time. As in "NVN" computers, each of these tasks can be considered a single process for a specific application, which in this case is the building of a house. However, the sufficient conditions must be present in order for a task to be performed concurrently. Allocating a roofer to shingle a house presumes that there is a roof to shingle. If the carpenters have not progressed to the point of building a roof, it make no difference whether or not a roofer has been assigned to shingle. Such is the case with "NVN" machines. If an algorithm has been decomposed into parallelizable processes, there must exist available processors in order to achieve the necessary speedup.

4.4.1.3. Synchronous vs. Asynchronous Algorithms. An important issue when designing a parallel algorithm is whether algorithms communicate between one another synchronously or asynchronously. In synchronous algorithms, tasks communicate interactively with other tasks, but must wait for other tasks to finish before proceeding. Also all of the parallel tasks must wait for the slowest task which reduces performance. However, if a problem can be evenly distributed where each processor performs the same instructions and communication scheme, processing is very fast. In asynchronous algorithms, tasks are not required to wait for each other, but rather communicate dynamically through messages or global memory. There is a need for much more overhead due to the nature of communication. Nevertheless, a wide range of application areas lean toward this implementation.

4.4.1.4. Speedup vs. Communication Costs. The primary reason to exploit concurrency in a given algorithm is to obtain an improvement in the speed of algorithm execution. Obviously, this is obtained via concurrent processing by executing several portions at the same time. However, a trade-off must be considered between speedup and communication costs when the communication complexity of a certain algorithm is higher than the computational complexity. Once again, consider the building of a house analogy. Suppose a number of masons are building a brick wall. Their individual responsibilities are to construct a wall using bricks and mortar supplied by an apprentice. If the apprentice cannot mix the mortar and/or supply the bricks to the masons fast enough, the masons will experience periods of inactivity and will not perform to their full capability. Such is the case with computers. For example, if more time is spent routing data among processors than actually performing the computation, little is gained.

4.4.1.4.1. Amdahl's Argument. At this juncture, it is important to reflect upon a measure of speedup that can be obtained. When referring to speedup calculations, Amdahl's work is usually the point of reference.

Amdahl's argument (a more formal definition is known as Amdahl's Law) states that for any fixed number of processors, speedup is usually an increasing function of problem size [17], [36]. Realistically, Amdahl's argument proposes that the limitations to speedup are due to the degree of parallelism in an algorithm. The limiting factor to increasing the speed of a concurrent algorithm is the reciprocal of the fraction of the computation that the concurrent algorithm must do sequentially. This is normally derived by the formula shown in Figure 4-3.

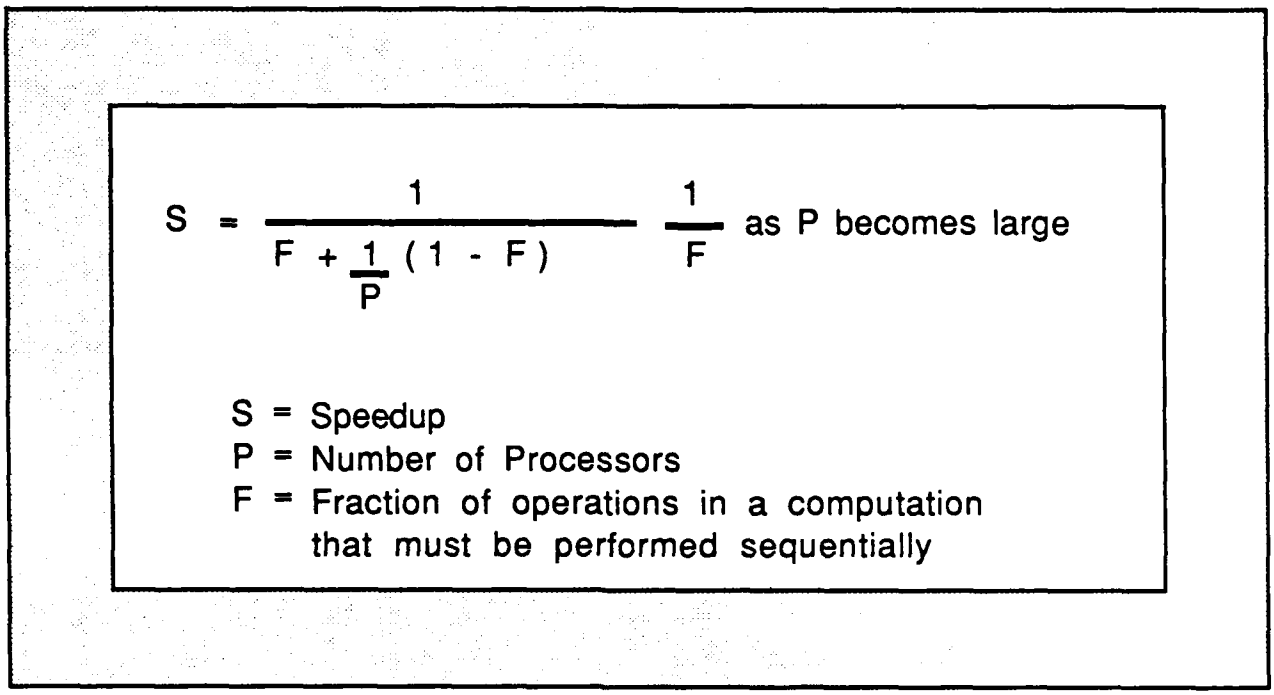

Figure 4-3: Amdahl's Argument

However, it has been observed that this argument relies on the estimation of the percentage of sequential operations in an algorithms execution. For example, a very significant speedup is realizable if the portion of the sequential instructions in a program is estimated to be very small, say one percent, as compared to ten percent, of the entire program.

4.4.1.5. Architecture Considerations. Ultimately, the rate of success or failure of parallel processing implementation directly depends upon the ability to implement an algorithm efficiently on a particular non-sequential system. The level of parallelism is also highly dependent on the machine architecture that will be utilized. Clearly there is a wide variety of machines and computers that can be classified as "NVN" which are possible candidates. As mentioned previously, these machines may range from array processors to vector processors to multiprocessors, all of which have distinct architectures. The multitude of machines

provide a number of diverse means to solving a particular problem via some parallel route. However, because of this, the performance of an algorithm can be radically different on different architectures. Many factors can account for this, such as communication overhead, synchronization, and granularity. The key point here is that the algorithm must be designed to fit a selected architecture.

4.5. Languages and Programming. A major software issue for "NVN" systems is that of partitioning a given application by extracting parallelism via computer programming techniques. This can be obtained either explicitly or implicitly. (Normally, explicit parallelism is provided by a programming language and implicit parallelism is provided by a compiler). Basically, explicit parallelism provides a user with programming abstractions that specify concurrent operations. However, it has been observed that conventional languages lack the syntax and vocabulary for specifying parallelism.

"NVN" programming should not be a totally new concept for most of today's experienced uniprocessor-based programmers who design and develop code for large-scale sequential machines. Basically, the underlying principles and fundamentals of von Neumann programming can be carried over into the "NVN" world. For example, many of the basic operations, like synchronous shared variable access methods and shared data structure concepts, are highly utilized by programmers involved in many of the sophisticated uniprocessor operating systems in current use (for example, the latest versions of MULTICS and Unix). However, "NVN" programming appears to be very excruciating for most of today's programmers.

When constructing a house, a builder has a number of factors to consider in order to implement many design features. For example, consider the exterior of a house. The builder can finish the exterior in a multitude of materials such as aluminum siding, vinyl siding, bricks, cedar planks, etc. Each has their advantages and disadvantages which range from price

to durability to appearance. There are also many factors to be considered in choosing a particular programming language for a "NVN" architecture. The most important factor of all is the application domain feature [4], [18]. For example, LISP is generally used only for artificial intelligence applications, the C programming language is preferred for operating system based applications, and Ada is usually better for data description applications [18], [39]. Obviously, there are numerous trade-offs between the use of one language over another. Therefore, when choosing a language, a careful study of the advantages and disadvantages of a particular programming language should be made in order to effectively compare it to other candidate languages. For non-von Neumann computers, certain language extensions are usually necessary to exploit parallelism [21]. Moreover, there are also a wide variety of implementations for a single programming language. These are also important considerations. If no specific language, or extended language, can be utilized to support a specific application, the development of a new programming language may be necessary.

There are basically two methods to high-level programming language development for "NVN" computers - new language development and existing language extension. From personal observation, it appears that the majority of the currently available parallel computer languages have been derived from one of the two methods, although some "NVN" machines utilize existing sequential languages and force the compiler to detect any inherent parallelism of a program. For example, this is especially true in vector processors. Basically, a vector compiler will compile a program written in a specific sequential language and attempt to extract any inherent parallelism to generate vector instructions. This section deals only with explicit language development (compilers are discussed in paragraph 4.6).

4.5.1. New Language Development. If no specific language, or extended language, can be utilized to support an application, the development of a new programming language may be necessary. The same holds true for a

carpenter. For example, if a carpenter cannot purchase or adapt a window for a specific application, he will build one from scratch. The new language approach is based upon developing new concurrent or parallel languages for the purpose of supporting non-von Neumann architectures. An analysis of the new languages that have been developed (such as Occam, PPL, Actus II, BLAZE, and Parallel Pascal to name a few of the more well-known languages) has shown that these languages usually contain various application oriented parallel constructs that are oriented toward a particular architecture. Therefore, these new languages are usually not very portable from one machine to another. Moreover, none of these new languages have been universally accepted in any commercial supercomputer.

Probably the most widely recognized new programming language that has been designed primarily for parallel processing is Occam. Basically, Occam provides explicit partitioning mechanisms where variables are declared by the programmer at the beginning of a program. However, Occam is not universally viewed as a sufficiently diverse parallel programming language since it does not support many applications.

4.5.2. Existing Language Extension. As is the case with developing parallel algorithms (as discussed in section 4.4), the utilization of an existing sequential language can be exploited. This method allows for an existing sequential language to be extended using "NVN" architecture constructs to support concurrent processing. For example, FORTRAN extensions have been implemented to control directed decomposition of specific program elements, such as large DO loops, by extending a new DO PARALLEL directive to the existing language [10]. Other program directives have also been extended to the standard FORTRAN syntax which establish how (and if) data will be shared among the individual processors of a "NVN" system. Other extensions of conventional FORTRAN have been utilized by only incorporating communication facilities that are added to specify message passing. Normally these extensions to existing serial languages are very dependent upon a

specific "NVN" architecture and are explicitly designed to utilize the extreme computing power of the specific architecture. Obviously, there is very little hope of portability since these languages are usually extended with one machine in mind. This causes two very critical issues. First, if a user employs a new target machine, recoding of a particular application will most likely need to be accomplished in another extended language, or by re-extending the originally employed language. Secondly, since the extended language is designed to exploit the underlying architecture of a particular "NVN" machine, the programmer is forced to understand the architecture. Thus, program development is much more difficult.

4.5.3. Obstacles. There are a number of direct observations that can be made of the obstacles confronting both new language development and existing language extension methods:

1) Learning a new programming language places many problems on the programmer. For one, programmers are very resistant to change and therefore tend to solve problems in terms of well-established and familiar language structures. Therefore, this makes concurrent programming more difficult right from the start. Also, learning programming details of a new language, or of an extended standard language, is very difficult.

2) Programming in a concurrent language is much more difficult than writing sequential code. Concurrent program logic is more difficult to understand and implement in code. Also, concurrent language syntax is different. The specification of processes and the control of their interactions is very important which is something that is usually not considered in sequential programming. However, some do not share this observation. Some believe that the parallel behavior of a sequential program need not be considered [25]. The position taken here is that concurrent processing is very different from normal sequential programming and therefore may require a better solution than just a new

language or language extension. Software support tools are lacking for concurrent design and implementation at the language, compiler, and run-time package levels.

3) Because of architectural differences, software reusability between different "NVN" machines is usually not possible. Obviously then, portability is almost non-existent unless the new target machine is inherently similar to the original machine. It is very difficult and time consuming to re-write code every time an application is hosted on another machine. Since experience gained from utilizing one programming environment is usually not transported to another environment, expensive software development costs are likely to incur. This may directly affect future gains in software productivity. If reusability problems can not be alleviated, at least to some degree, "NVN" use in "mainstream" computing will be greatly hindered.

4) Program modification or maintenance is virtually impossible by anyone other than the original programmer. By the very nature of parallel programming, one programmer will normally code in a manner that is unique. This is based on their background in computing/programming, understanding and interpretation of the programming language, and understanding of the host computer.

5) Code designed and written for a "NVN" machine may be extremely difficult to debug. Since deadlocks may occur due to the very nature of the architecture which the programmer is unaware of, bugs are not necessarily due to errors in the code. Also, since operations on multiple processors do not necessarily occur in precisely the same sequence from one program execution to another, finding exactly where errors occur can also be very difficult.

4.5.4. Ada. Since the development of the Ada programming language was sponsored by the United States Government to write effective software for computer components of military systems, particularly for command

and control systems, it is a strong programming language candidate for "NVN" architectures in the DOD world. The very nature of many DOD systems underly many of the principles of parallelism. For example, typical military command and control systems often require the monitoring of many concurrent activities, whether they be worldwide or systemwide. The ability to keep track of and monitor many concurrent activities and quickly respond to unpredictable events, is a very important concern of command and control systems. Writing effective software for these systems is of the utmost concern. Nonetheless, it is very unclear as to whether Ada is a good choice for concurrent software development.

As alluded to above, one of the reasons that Ada was specifically designed was to facilitate concurrent programming for command and control systems. Most of today's popular programming languages were designed for writing sequential programs. Ada, however, allows a programmer to establish many separate threads of control via multitasking. The trends toward multiprocessing and distributed systems dictates the need for concurrent programming languages like Ada. However, up to this point Ada has not been widely accepted in the sequential world, much less in the non-sequential world of "NVN" architectures. This is due mainly to the lack and immaturity of Ada support tools and inefficient real-time processing capabilities [18]. Therefore, it is very uncertain whether Ada will play a major role in language development for "NVN" architectures. A serious challenge exists to understand and extend the benefits that Ada possesses for "NVN" applications.

4.6. Compilers. There is also a very urgent need for the development of compilers for the successful utilization of "NVN" architectures [7], [33]. Parallelizing compilers allow for implicit extraction of parallelism. Basically, the compiler uses program restructuring techniques to transform a sequential program into a concurrent form that is suitable for "NVN" machines. There are primarily two methods for compiler

development, namely new compiler development and existing compiler extension. These are further discussed in the following subsections. It should also be noted that although there exist a number of automatic parallelizing compilers in the marketplace, most of these only achieve a very small degree of parallelism and are usually limited to small, localized segments of code.

4.6.1. New Compiler Development Method. In the case of new parallel language development, there is a very direct correlation between the constructs and capabilities of the newly developed compiler to the underlying architecture. Again, compiler portability is a major concern. However, it is not the concern of the programmer, but rather of the writer of the compiler. At this point in time there does not exist a universal compiler that can be utilized for even a subset of "NVN" architectures. More effort needs to be directed for new compiler development as well as to new language development.

4.6.2. Existing Compiler Extension Method. The underlying premise of this method is based upon the utilization of existing serial compilers already in the marketplace. It should be noted that there is an obvious overtone of the hand-in-hand association between languages and compilers. Therefore, this method is basically an extension to the existing language extension method (section 4.5.2). This method is the more widely accepted and used of the two, but it has its advantages and disadvantages:

4.6.2.1. Advantages. One advantage for the use of an existing language and its associated compiler allows for programming in a familiar language where no new constructs need to be learned. A second advantage, utilizing existing compilers, a programmer can design, write, and debug new code for parallel algorithms in the same manner that they are accustomed to for sequential processing. A third advantage, utilizing an existing serial compiler and language, the programmer can utilize existing sequential programs and applications to be transposed

directly for parallel applications. The fourth, and last advantage, is the issue of portability. Usually only a minimum of changes are required to the existing compiler (to support changes made to the existing sequential language) when shifting an application to a new machine.

4.6.2.2. Disadvantages. All the inherent concurrent constructs of a particular program may not be recognizable by its accompanying compiler. Secondly, a program cannot be written in which the compiler can take full advantage of the machine and its underlying architecture without a great deal of effort. This would require the programmer to be intimately familiar with the architecture at hand as well as assist the compiler, through sophisticated programming techniques and tricks, to recognize and exploit the architecture. Another disadvantage is that the use of a sequential compiler and its associated sequential language, forces a programmer to program parallel algorithms in sequential form.

4.7. Operating Systems. Operating system issues for "NVN" computers are similar to those for large computer systems that utilize multiprogramming [34]. In a typical multiprogramming system, the operating system will allow for more than a single program to be in some state of execution at the same time. This is similar, from an overall standpoint, to the way an operating system for a "NVN" based system should work. However, there is a very big difference in the complexity of a"NVN" operating system and a multiprogramming operating system. A "NVN" operating system must support a number of asynchronous tasks which are running in parallel via the simultaneous interaction of multiple processors, not multiple programs.

"NVN" operating systems must also have capabilities that are beyond the scope of those required for a multiprogrammed system, such as additional I/O capabilities. Resource utilization, processor load-balancing, and system reconfiguration problems must now also be handled in addition to the exception handling, system deadlock, memory contention, resource allocation, data protection, and management overhead problems that are

supported in multiprogramming systems. For resource utilization to occur, a mechanism must provide the necessary vehicle to assign a task to each available processing node (or set of nodes) as the tasks and nodes become available. In processor load-balancing, the ability to effectively distribute tasks among the processing nodes in order that maximum utilization occurs is the key concern. System reconfiguration issues are also very important. Basically a fault-tolerant concern to insure that a failure does not "kill" a system, system reconfiguration allows for graceful degradation of an affected system where the only outcome is lessened performance.

Typically, the vast majority of industrial and academic endeavors in the development of operating systems for "NVN" architectures has resulted in either front-end hosts or in adaptations of existing operating systems. However, many of the operating systems that run on "NVN" systems are limited to library routines with very little additional capabilities. Many "NVN" computers themselves are only back-end machines that utilize front-end host machines to provide many of the functions that the necessary operating system would provide, such as I/O support and processor scheduling. Most are built upon either a version of the UNIX operating system, or upon Mach, which is a new experimental distributed operating system aimed at parallel processing systems utilizing multiprocessors (built by researchers at Carnegie-Mellon University and funded by DARPA). Mach overcomes many of the limitations UNIX places on "NVN" system configurations by implementing message passing and shared-memory techniques. However, Mach also has many disadvantages. UNIX has been widely used because there exists a large, well-established user community. Many programmers are quite familiar with UNIX. Also, many UNIX facilities do not have to be reinvented for "NVN" machines since some are applicable to both the von Neumann and non-von Neumann world of processing. However, UNIX does possess a multitude of limitations, such as user interface deficiencies and interprocessor communication problems, which need to be resolved for any widespread acceptance in many current advanced systems [37].

More emphasis needs to be placed on developing new operating systems for "NVN" architectures, other than just expanding UNIX or Mach, which seems to be the current "rule-of-thumb." There are currently only a few schemes that have been employed in the design of new operating systems for "NVN" architectures. These schemes have been reluctantly used, which can be attributed to the widespread use of UNIX and Mach adaptations. However, there exist a great number of deficiencies peculiar to many "NVN" systems that these approaches do not support. The three schemes presented below offer a glimpse of some of the work that is being accomplished for "NVN" operating systems development. More endeavors in this area need to be pursued, but the three schemes presented here are a step in the right direction. It is felt that all three offer insight into many of the issues confronting "NVN" operating system development which will hopefully lead to something that will significantly advance the state-of-the-art.

4.7.1. Master-Slave Scheme. In this operating system configuration, one of the processors of a "NVN" machine is given the role of the executive supervisor (master) for the purpose of allocating work to the other processors (slaves), as well as maintaining the status of them. The executive program always runs in the master processor, This scheme is subject to catastrophic system failure if the master processor fails. This operating system scheme is very effective for systems that employ an asymmetrical architecture configuration. Figure 4-4 illustrates this scheme.

Overall, the master-slave operating scheme is relatively inflexible when compared to the other two schemes.

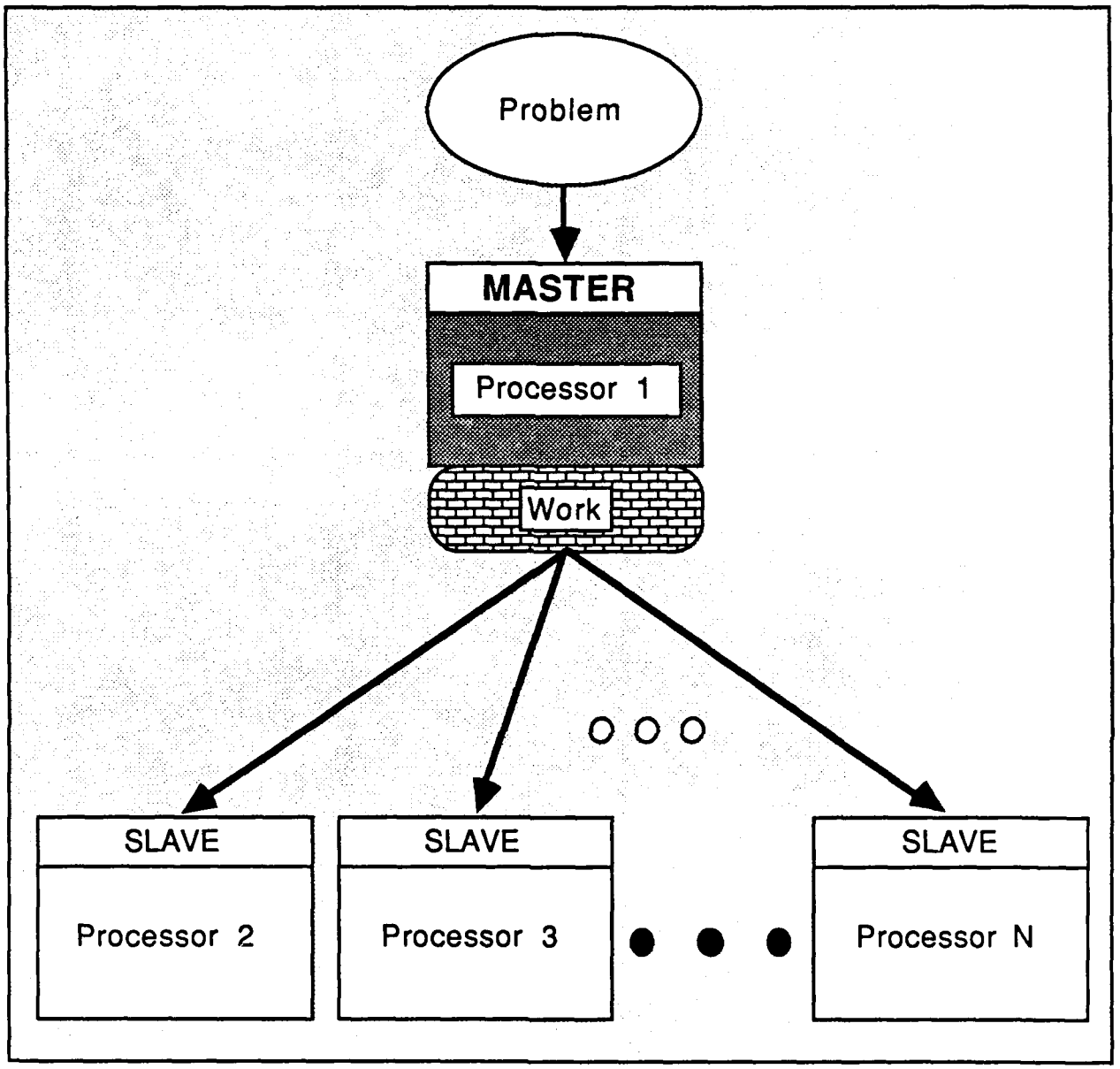

Figure 4-4: Master-Slave Operating System Scheme

4.7.2. Separate-Supervisor Scheme. The separate-supervisor scheme, portrayed in Figure 4-5, allocates control to each processor in a "NVN" system. By having a separate supervisor in each processor, each

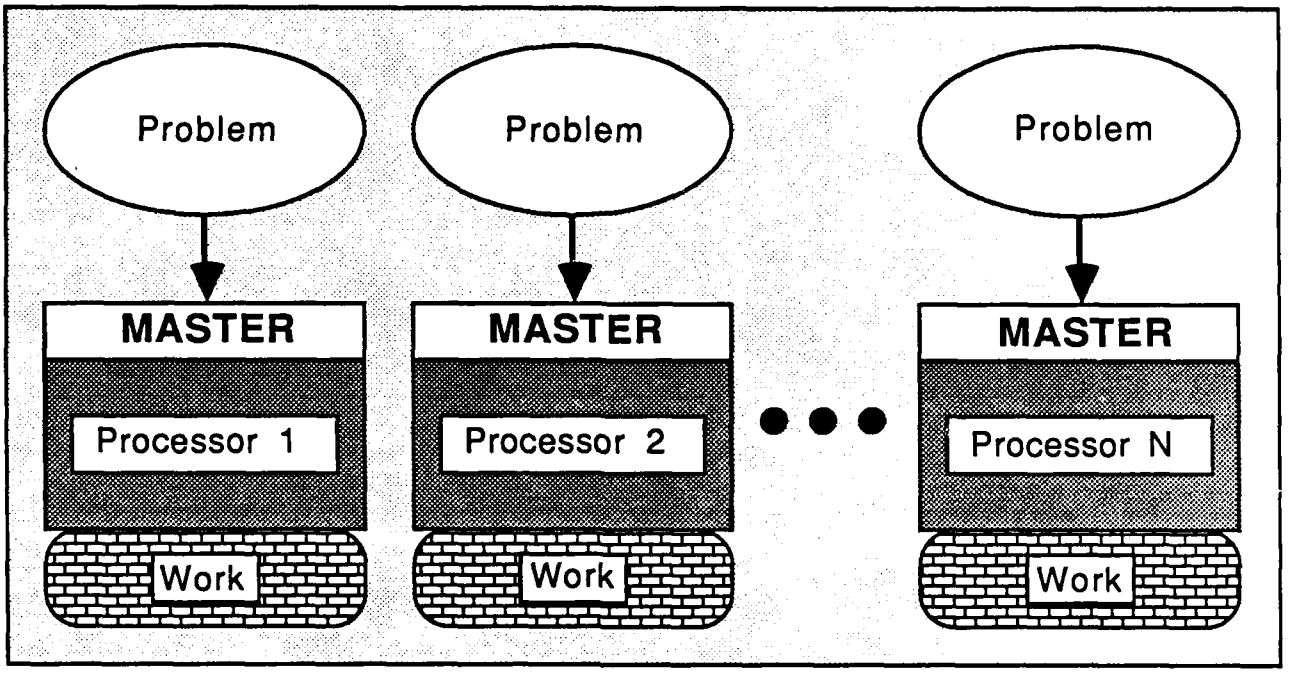

Figure 4-5: Separate-Supervisor Operating System Scheme

processor services its own needs and has, in effect, its own set of I/O, files, data, etc. The separate supervisor that exists in each processor is basically a kernel, or subset, of the operating systems.

4.7.3. Floating-Supervisor Scheme. This scheme employs a method of "floating" the supervisor role (master) from one processor to another. This method allows for the greatest amount of flexibility and greatest resource efficiency, but is also the most difficult to implement. Another critical issue exists in the very nature of the scheme - more than one processor can execute supervisor functions at the same time - which can obviously create havoc. For example, since several processors can be in a supervisory state simultaneously, access conflicts can occur. This scheme does however provide for very graceful degradation in the event of a failure within a system. This scheme is shown in Figure 4-6.

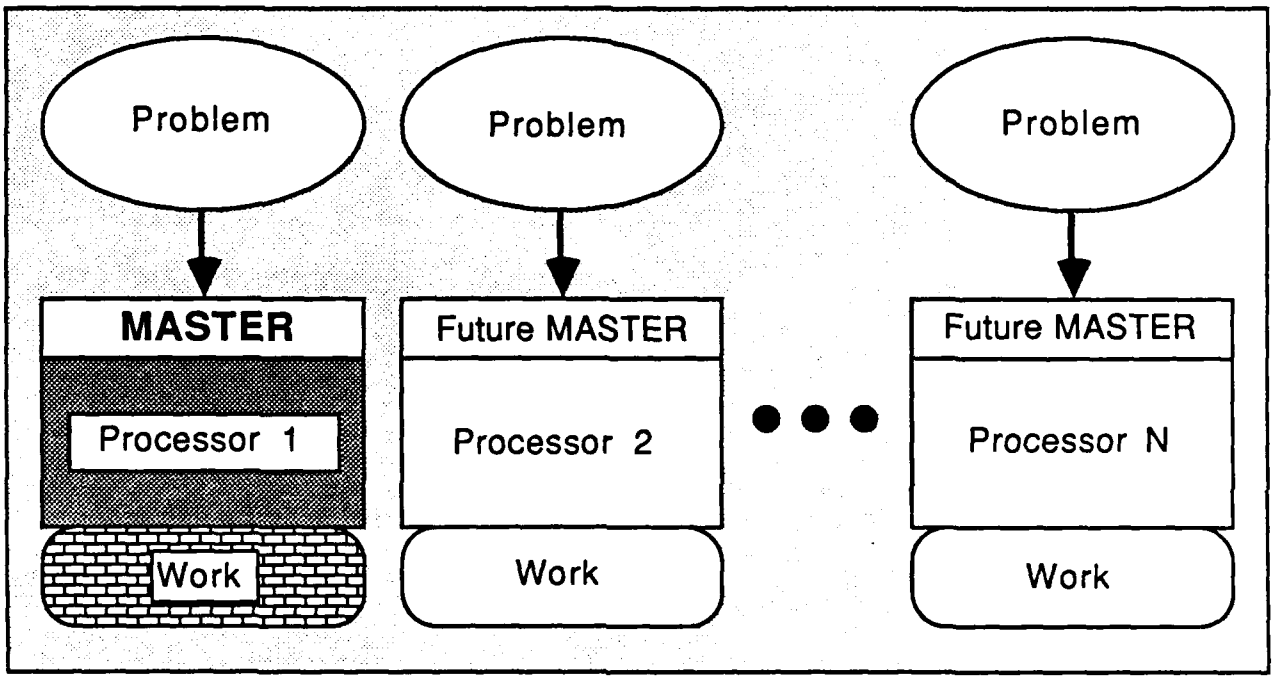

Figure 4-6: Floating-Supervisor Operating System Scheme

4.8. Fault-Tolerance. Fault-tolerant techniques must be understood and utilized for the successful utilization of "NVN" systems as well [3], [22], [25], [44]. In any system, whether it be von Neumann or non-von Neumann in nature, a number of steps are required to handle faults:

1) Detection. Detection is the realization that a fault has occurred based upon an observation of the symptoms that are present.

2) Diagnosis. Diagnosis provides the necessary method or methods to ensure further contamination of the system does not occur. This is accomplished by the identification of the underlying causes of a fault.

3) Recovery. Recovery provides the necessary action to retain and preserve the availability of a system.

Hence, a "NVN" system must be capable to automatically detect and isolate failures, and to dynamically reconfigure itself in an environment that may possess multiple errors. Any single failure should not be catastrophic, but rather, result in a graceful degradation of system performance. Generally, detection is accomplished by "spying" on a processor, usually involving duplication of hardware modules. Results are normally decided upon via some kind of voting mechanism. There are a number of varied techniques that employ this philosophy. However, there are a number of significant disadvantages. For example, a large amount of extra hardware is required and graceful degradation is not usually possible. Dynamic reconfiguration is controlled by hardware and/or software. Today's systems possess memory configurations that usually include error correcting and detecting schemes. Also, most local area networks have provisions to continue operation in the presence of errors. Efforts are needed to provide both hardware and software fault-tolerance for "NVN" architectures. Error checking throughout a "NVN" system must be more active and more comprehensive than in a sequential environment. *Limiting the propagation* of errors is also more critical.

Many "NVN" architectural topologies inherently possess many of the necessary underlying vehicles to easily facilitate fault-tolerance. For example, consider the Hypercube. The very nature of its physical interconnections allow for more than one optimal configuration for a number of applications. This is due to the fact that there exist some applications that do not utilize the total topology of a Hypercube. Consider a 32-node Hypercube configuration that supports an application that only uses an 8-node processing scheme. Since there exist many 8-node configurations, if a node "dies" the application can be mapped to another 8-node configuration which offers the same performance since it is just as optimal as the original configuration. Obviously, hardware redundancy and interprocessor communication are very important in a Hypercube scheme.

4.8.1. Hardware Fault-Tolerance. As alluded to above, hardware fault-tolerance is typically characterized by brute force methods to assure high system reliability and availability. Basically, hardware fault-tolerance assumes that the hardware works correctly unless there exists a transient or solid component failure. Usually hardware fault-tolerance is achieved through hardware redundancy and comparison of outputs of parallel processors.

4.8.2. Software Fault-Tolerance. Software fault-tolerance can be simply defined as the reaction and accommodation of faults that occur during the execution of a program. Typically, research into the field of reliable software development for conventional von Neumann computers has evolved from the adaptation of hardware reliability techniques to software. However, software fault-tolerant development techniques are much more difficult than hardware fault-tolerance mechanisms. Work is needed to address software fault-tolerance and its contributions to meeting "NVN" system requirements. Software for non-von Neumann computers is intrinsically error prone due to many factors. For example, the nature of multiple path execution can place a number of different errors throughout a process.

The two most widely employed schemes for providing software fault-tolerance are the recovery block scheme and the N-version programming scheme. The recovery block scheme incorporates strategies that provide for error detection, backward error recovery, and fault treatment. N-version programming uses N independently designed modules that work concurrently to "vote" on outputs. For computer systems that employ limited hardware resources, or possess characteristics which make a voting check inappropriate, the recovery block scheme is usually chosen. However, for systems that are comprised of redundant hardware resources, the N-version programming technique is more desirable.

Fault avoidance is also a key issue of software fault-tolerance. To accomplish this, the characterization and development of new and innovative software requirements and design strategies that "build-in" software fault-tolerance are needed for "NVN" architectures. New programming and testing techniques that verify software fault-tolerant performance are also of importance, possibly utilizing automated test tools and proof of correctness techniques.

5. SOFTWARE UTILIZATION ISSUES

5.1. Basic Considerations. Effective software utilization issues for "NVN" architectures are the next logical venture after software development has been accomplished. Issues include software development methodologies (and the impact of DOD-STD-2167A), programming environments, software tools, advanced communication methods, and software development environments for hybrid systems. The following sections are intended to provide a better understanding of some of the issues associated with the need for these software utilization mechanisms for the present and future use of "NVN" systems.

5.2. Software Development Methodology and DOD-STD-2167A. An assessment is needed to identify the various shortfalls within the current set of software engineering methodologies imposed upon by "NVN" architectures. The specific phases of the software life cycle and how the life cycle is affected by developing software for "NVN" machines must also be considered as well. The software development methodology most familiar to military software development is DOD-STD-2167A (and its predecessor, DOD-STD-2167) [12]. Because of this, this section deals primarily with DOD-STD-2167A and not to other similar methodologies.

In the 1970's, the "waterfall" model of software development was devised to help alleviate the problems that were being encountered in major system development efforts. The waterfall life cycle model is primarily based upon setting firm requirements early in the software development

process to avoid the extremely high cost of making requirements changes to nearly completed software. DOD-STD-2167A is based upon the waterfall model.

Issued by the Department of Defense, DOD-STD-2167A is a specification standard that military software systems must conform to. DOD-STD-2167A is well known to most everyone involved at all in military software contracting. in either government, industry, or academia. Basically, this standard was developed to establish a uniform set of requirements for software development that are applicable throughout the system life cycle, namely through the acquisition, development, and support of software systems. Specifically, six phases of the software life cycle are identified in DOD-STD-2167A: requirements, preliminary design, detailed design, coding and unit testing, integration, and system testing. Each phase must be analyzed to determine what impact "NVN" machines and "NVN" applications place on each phase and what changes (if any) are required. For example, consider the coding and unit testing phase. In this phase the software engineer is required to produce an implementation of specification modules in a chosen language and then perform unit testing. However, some new problems have surfaced due to the advent of "NVN" architectures. For one, there exists a lack of debugging tools to assist the software engineer. Also, memory contention bugs and deadlock bugs are more difficult to detect. Some "NVN" machines (mostly distributed memory machines) do not allow for any I/O from the nodes. Furthermore, there also exists a data display problem due to the fact that most "NVN" computer perform so rapidly that the large amount of data that is produced cannot be viewed in real-time. Hence, because of problems such as these, simulation techniques have been devised to assist the software engineer prior to the commitment to a particular "NVN" system. However, simulators do not exist for many machines. A decision must then be made as to whether or not a simulation should or should not be developed first. It is felt that the development of a simulation would be very beneficial.

As alluded to above, there exist many definitions and variations of the software life cycle, but are all made up of essentially the same phases. Figure 5-1 portrays the system and software life cycle as seen from the perspective of DOD-STD-2167A.

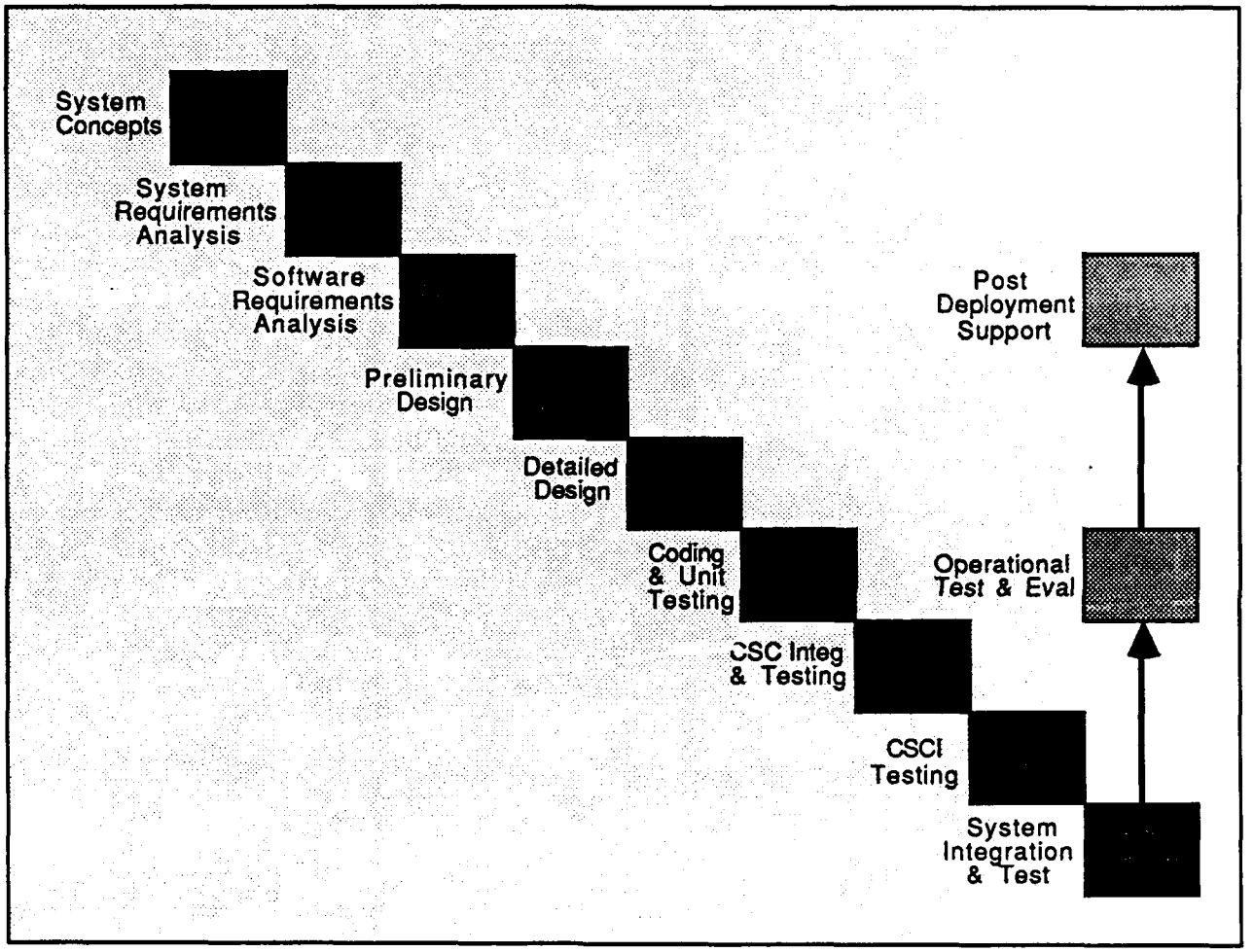

Figure 5-1: The System and Software Life Cycle

However, this software life cycle, and its cousins, were developed when most software was designed for sequential-based von Neumann architectures. Because non-von Neumann architectures execute primarily in a non-sequential domain, a number of previously undefined issues need to be included in the software engineering

methodologies that are to be used for software development for this class of architectures, such as:

1) the underlying architecture (hypercube, multiprocessor, data flow),

2) the programming technique that is employed (parallel language, extended serial language, vectorizing),

3) problem decomposition issues,

4) interprocessor communication problems,

5) memory contention issues,

6) I/O capabilities, and

7) how to accommodate high data volume.

Therefore questions arise concerning the specific nature of the software life cycle for "NVN" architectures since additional issues may or may not lead to a refinement or expansion of the phases within the life cycle.

Generally, the overall structure of the software life cycle depicted in Figure 5-1 can be applied to "NVN" architectures with minor changes. However, at this point it is very unclear as to the nature of possible refinements to the specific phases of the software life cycle. These changes are direct modifications that support new software development issues that have surfaced with the advent of "NVN" processing. The common thread of this report has repeatedly stressed that it is more difficult to design and develop software for "NVN" systems than for von Neumann systems. *Software development issues for "NVN" computers* introduce such concerns as communication overhead, synchronization, decomposition, and memory contention problems that were previously

not considered for von Neumann machines. For example, consider the product design phase of the software life cycle. Before any detailed design of a system is conducted, it is necessary to decide on a particular mode of parallelism. This is primarily due to:

1) the large number of parallel modes (SIMD, MIMD, vector, etc) that are currently available,

2) the various programming methods that exist, and

3) the wide variety of interconnection networks (bus mesh, crossbar, hypercube, etc) that are currently being employed.

5.3. Programming Environments. A new paradigm in programming systems has evolved over the last few years, known as the programming environment [1], [16], [37]. It is considered a major software issue for the effective use of "NVN" machines. A programming environment designed with "NVN" architectures in mind is usually based upon the same framework for serial architectures but with a different set of requirements. Basically, a programming environment can be defined as a collection of software engineering tools and techniques capable of supporting software life cycle development that should allow programming on a "NVN" machine to be fairly easy and straightforward. Such an environment oversees all of the activities related to the development of a computer program. Typically, these tools include language-oriented editors, debugging tools, program management facilities, monitoring and instrumentation tools, and performance enhancement mechanisms.

The advent of "NVN" machines has introduced new issues that must be considered in the design of a programming environment. Programming environments are built to provide a number of tools for both assisting the programmer as well as helping to manage programming. The hardware that comprises the "NVN" system is very important as well, because it

limits the type and functionality of the tools that can be supported in a "NVN" environment. Probably the most important consideration when designing a programming environment, whether it be for a von Neumann or non-von Neumann based system, is integration. The collection of tools that form the programming environment must interact and support one another. Therefore, the method employed to achieve this integration is also a very important consideration.

It is apparent that with the advent of "NVN" computing, parallelism must be supported both in future languages as well as in their programming environments. This is true for all facets of non-von Neumann computing, whether it be for a single machine or a network of machines. Programming many processors with a diverse multitude of communication and synchronization occurring between them is very difficult. A programmer needs a level of abstraction that allows parallel tasks, and their associated data exchanges, to be accomplished as simply as possible. For example, a user should not have to specify how a set of tasks should be allocated. Furthermore, how they are implemented should also not have to be specified. The compiler and operating system should provide the required facilities to alleviate the specifying of these things, as well as for low-level concurrency and message routing. A sophisticated debugger should also be available to detect and isolate errors. As mentioned previously, the debugger should be capable of analyzing results, diagnose errors, and respond to users.

In general, the programming support environment should supply the underlying framework for effective "NVN" implementation. It should be designed and integrated so that a user can apply an application concept and then carry out the necessary design in a simple and straightforward manner. At this point in time, it does not appear that there are any efforts that are significantly addressing these problems. Perhaps the Software Life Cycle Support Environment (SLCSE) could be adapted and extended to provide such support in the future.

5.4. Software Tools. In order for "NVN" machines to be utilized to their full potential, various software engineering tools are needed. It is therefore very important to obtain information about the software engineering tools, techniques, and methods that are currently available for use on "NVN" architectures in order to determine how they can be exploited for meeting the needs of many applications, including C3I applications. Although many software engineering tools have been developed for von Neumann machines, it is much more difficult to develop software and software tools that take full advantage of the processing power associated with "NVN" machines. The insufficient fallout of software that has occurred has drastically resulted in decreased productivity for software developers.

Software engineering tools are needed to examine and understand program performance and operation in concurrent environments. These tools can be designed to assist both the programmer and program manager at the system level. These include parallelization type tools, software translation tools, program restructuring tools, debuggers, analysis tools, simulators, linkers, loaders, application libraries, software performance monitors, software directories, management tools, monitoring tools, control tools, automated data collection tools, and windowing methods to name a few [32]. There are currently well over a hundred software engineering tools available in the marketplace for "NVN" computers, some of which are depicted in Figure 5-2. The point here is that, while there does exist a fair number of tools, there is a growing need for even more sophisticated tools that break away from the normal development methods that go into the design of software engineering tools. The implicit behavior of "NVN" systems, and their capabilities, need to be further exploited into the realm of software support tools.

Tool Name	Type of Tool	Language/System Support
SeeCube	Monitor	NCube
Monit	Performance Monitor	Sun-3 Workstation
Belvedere	Debugger	Parallel Languages
Euclid	Simulation Model	Multiprocessing
Parallel Fortran Converter	Translator	Fortran
Parafrase	Program Restructuring	Fortran, C
Pdbx	Debugger	Sequent
Instant Replay	Debugging Environment	BBN Butterfly
SCHEDULE	Program Interface	VAX 11/780, Alliant FX/8, Cray-2

Figure 5-2: Some Existing "NVN" Software Tools

As alluded to above, most of the software engineering tools and techniques that are employed today for "NVN" machines are nothing more than straightforward extensions of traditional software development methods previously designed for serial computer systems. For many applications, these tools are inadequate or inappropriate for the successful software design development and

maintenance issues unique to "NVN" systems. For example, consider debuggers. It is easy to see that parallel programs are much more difficult to debug than sequential programs. After a bug is supposedly fixed, it may be impossible to reconstruct the sequence of events that led to the exposure of the bug in the first place. Obviously, it is also not feasible to correctly certify that a bug has been removed. For example, debugging on the Hypercube is generally accomplished by receiving and responding to actions that a user specifies, such as breakpoints or by displaying memory or register contents. The unique capabilities and applications supported by "NVN" architectures need to be reflected in the tools, techniques, and methods used throughout the entire software life cycle. Some of these tools need to be designed to provide a user with the necessary capability to deal explicitly with the non-sequential nature of "NVN" systems. Others need to be designed to permit a user to design software targeted for sequential machines and to then employ the necessary tools to effectively exploit the concurrency of the system and express it automatically.

5.5. Communication/Synchronization Methods. In order for "NVN" machines to effectively operate, a variety of communication and control mechanisms to minimize contention, and to optimize use of system resources, are needed. To accommodate communication and synchronization among processors, a particular interconnection network (as portrayed in section 3.2) must be implemented. Deciding on a particular interconnection design choice is normally the result of many concerns, such as the application demands for the machine, the number of processors that will be employed, and the data speed requirements to name a few. A number of technical questions must also be imposed by a system designer prior to the selection of one interconnection network over another, such as:

1) What type of switching methodology should be employed? Basically, there are three switching methodologies - circuit, packet, and integrated. In circuit switching there exists a physical path between

two nodes. In packet switching there does not exist a physical path. Rather, data is stored in a "packet" (with a prespecified length) and then independently routed to a particular destination, usually using store-and-forward procedures. Integrated switching provides the capabilities of both circuit and packet switching.

2) What type of communication scheme will be present? Synchronous or Asynchronous? Synchronous communication allows for communication to be accomplished via a shared variable scheme or through message passing in which the connection paths are all centrally supervised. The particular architecture of a machine may dictate the type of communication scheme [16]. For example, consider multiprocessors. By definition, multiprocessors do not have any type of shared memory whatsoever. Obviously then, multiprocessors must achieve communication through means of message passing. Asynchronous communication provides for a dynamic communication scheme in which a communication path can be connected or disconnected at any time.

3) Should a static or dynamic network topology be used? A static topology allows for only dedicated links between processors wherein a dynamic topology allows for reconfiguration of the processors for whatever purpose, be it a processor failure or rescheduling of processors.

4) What routing technique is best suited for determining the available paths and resolving possible conflicts inflicted by path contention? Three methods are currently in use - centralized, distributed, and adaptive. These are pretty much well understood and widely utilized.

Synchronization is very much related to communication requirements. Basically, synchronization controls interference between the outside world and the individual processors. Synchronization is also concerned with the governing and cooperation of events in a system. For example, consider requests to and from memory for data. Clearly two individual

processes cannot access the same data at the same time. Synchronization mechanisms prevent this from becoming a deadlock situation.

Communication and synchronization methods are very important considerations for "NVN" architectures. Some are quite similar to conventional serial processing techniques, but the complexity of the topologies that comprise the architectures of many of today's advanced machines introduces many new issues and complexities.

5.6. Software Development Environments for Hybrid Systems. There also exists a need for the development of software when the problem domain entails the utilization of both von Neumann as well as "NVN" systems in conjunction with one another. A "hybrid" system of this nature is very likely when considering possible utilizations for military system configurations, especially for command and control functions, since specific functions can often be divided into distinct architectural classes (von Neumann or non-von Neumann). For example, a von Neumann class machine may act as a front-end scheduler for a "NVN" machine. Therefore, the development of software for systems that possess this configuration is also needed. Of course, the mix of existing serial based software tools would be a logical starting point for this development, but the "NVN" portion of the system, even if it comprises only five percent of the whole system, may be the bottleneck to overall system performance. Therefore, software development for the "NVN" portion is probably the critical area of concern for software development for hybrid systems.

Obviously, for the efficient software development in hybrid systems, the use of consistent development methods and transparent user support are needed. Software must be developed in such a fashion as to insure software engineering support is sufficiently powerful and flexible to adapt to the mix in architectures. Hybrid systems may possibly place constraints on the type of support that is required as well. Moreover, there may exist a need for dual environments, one for the "NVN" based

subsystem, and one for the von Neumann subsystem. System function allocation between hybrid classes, as well as programming techniques, are also important issues confronting the development of software for hybrid systems.

6. RECOMMENDATIONS.

6.1. Fundamental Research. To begin with, a number of issues need to be addressed at the fundamental research level. In general, there are two classifications of fundamental research that needs to be addressed, core research and applications research:

6.1.1. Core Research. Core research is needed to evaluate the use and improvement of "NVN" computing in the generic sense. Work needs to be done for algorithm development, the development of new software and new programs, and a baseline needs to be defined for analyzing which problems are best suited for parallel processing.

6.1.2. Applications Research. Applications research relates to specific areas in science and engineering which can utilize "NVN" computing. Work needs to be accomplished to determine how "NVN" computing can be exploited for such areas as artificial intelligence, logic programming, image processing, signal processing, and data processing, to name just a few.

6.2. Standardized Classification Scheme. As pointed out in section 3.3, the development of an adequate means of identifying and classifying the types of non-von Neumann and "NVN" architectures and implementations is sorely needed. The increasing number of new and diverse computer architectures and implementations for "NVN" systems are proof that parallelism is, and will be, exploited in the near future. Therefore, of significant importance is the need to classify these new architectures and machines as well as to obtain the necessary metrics to compare them for such things as performance, application domain, and

behavior. By doing this, a tangible criteria baseline can be developed to compare diverse systems so that a set of measurable objectives for possible applications can be made. Since a single application may possibly be solvable via a number of different "NVN" architectures, it is very important to gain comparable knowledge of the advantages/disadvantages of each system so that it can be more easily discernable why one architecture or machine is better than another. Obviously, the selection of the wrong architecture can preclude the use of a less effective and/or more costly system for years.

The recent activity in the research community, both in industry and academia, that has been directed toward the classification of "NVN" computers has resulted in either incomplete classifications, or in methods of classifying machines by examining only one architectural feature. For example, classification schemes have been devised based upon classification by: memory organization, control type, number of processors, processor size, synchronization overhead, and processing element assignments.

There exists a growing need for a more comprehensive method of identifying the fundamental characteristics of "NVN" computers, as well as identifying the applications they are best suited for. Possibly a hybrid scheme could be developed. The multitude of existing classification schemes has provided insight into the many different architectural features of "NVN" machines. Many possess information to identify the advantages and disadvantages that these machines offer in solving a particular problem. Primarily, the main consideration confronting the choice of a particular machine for a given application is how to best match the most important features of a particular machine to a specific application. However, how the other features of the machine may adversely affect the task. These "other" features may do more harm than good.

In the development of a hybrid classification scheme, it will be very important to include information that is contained in all of the previous means of classification, both the well known schemes as well as the not so well known academic-based type schemes, into one comprehensive scheme. Once developed, this architectural classification scheme could then be applied to existing machines as well as to models or simulations of proposed architectures.

6.3. Software Support. As discussed throughout section 5, many of the existing software practices that are currently utilized for sequential machines need to be readdressed for "NVN" based machines. "NVN" hardware advancements have progressed significantly in recent years, but "NVN" software development has not. In fact, software development for "NVN" architectures has not advanced very far from its uniprocessor origins. Programming languages, particularly high-level languages, are very limited in terms of support for the growing multitude of advanced computer architectures in existence today. The underlying concurrency applications that are supported by most of these architectures are currently not well suited for today's modern programming languages. New languages are needed. Still, most of the high-order programming languages that are being currently utilized to program "NVN" machines are only extensions of languages which were specifically designed for sequential machines. Moreover, these languages were primarily developed in the 1950's and 60's.

Multitasking, dynamic allocation techniques, dynamic load-balancing, task migration methodologies, and fault-tolerance are just some of the software issues that need to be integrated into an operating system that is capable of responding to uncertain and changing environments. The operating system must be able to assign, monitor, and terminate new or reallocated tasks. All this must be accomplished with real-time requirements and utilize processor resources optimally.

There is also a very limited set of software tools for "NVN" architectures [29]. Presently, there exists very few tools for the purpose of creating software programs for these new machines, fewer tools for debugging and analyzing these programs, and even fewer programming environments. A study of the tools that do exist, as well as an identification of those that are needed, is critically important if applications intended for concurrency are to be matched to these machines for C3I problems. Without sufficient knowledge of these tools, it will be impossible to effectively exploit what is currently available. Without insight into new tool development, potential uses of "NVN" machines is impossible as well. For economical reasons, whether they be financial, technical, or strategic in nature, the future of "NVN" systems is largely dependent on the success of software and system development.

Much more research is needed not only for the development of concurrent software algorithms and in software implementation mechanisms, but also in making them partitionable and fault-tolerant. Advances in programming support environments are also needed. There exist very few tools to assist a user in partitioning and evaluating a particular application to exploit any inherent parallelism. New languages and compilers are needed to extract parallelism. More specifically, different types of programming languages (conventional, object-oriented, logical, functional, etc) should be considered in order to determine the best concepts and building blocks to design effective parallel programs.

6.4. New Software Development Methodologies. New software development methodologies will most likely play a very important role in the effective production and maintenance of software for "NVN" computers. Whether or not DOD-STD-2167A can be utilized or be modified to assist in the development of a software development methodology is uncertain. The issues confronting the use of DOD-STD-2167A were previously discussed in paragraph 5.2 and are certainly valid for new development methodologies. A comprehensive comparison to the DOD-STD-2167A software life cycle needs to be accomplished to assess the

advantages/disadvantages and the strengths/weaknesses of new software development methodologies for "NVN" software production. Associated with this includes the development of new tools and techniques that will support a possibly new DOD-STD-2167A based methodology, with a complete analysis and examination of each phase of the life cycle.

Many new methodologies are emerging with the premise of "NVN" architectures as the root. Many are dissimilar to the typical "waterfall" approach employed in many conventional methodologies. One of these new approaches is rapid prototyping, which has received a great deal of attention. Basically, rapid prototyping utilizes the first part of the development cycle to build a quick prototype in order to analyze requirements, validate requirements, and to detect performance bottlenecks. In some paradigms the prototype is then discarded since it is never intended to be anything more than a means to check the design against requirements. This is based upon the observation that the requirements definition phase and subsequent system concept design phase (of the waterfall model) may occur years before the final implementation of the software. This assumes that the "right" software design decisions were made up front, since it places an early commitment to requirements for the final implementation of the software.

New techniques need to be developed that allow rapid, low-cost prototyping of existing, and/or experimental, high performance computers that are "NVN" in nature. Hopefully, a rapid prototyping mechanism intended for "NVN" computing could be developed that would provide the capability to evaluate alternative approaches to parallel processing applications such as developing simulations. By quickly building prototypes and running experiments, a number of design problems and performance bottlenecks could be identified early in the software design phase. Obviously, the major premise of this approach is the ability to quickly advance to each new iteration of the prototype and specification. Work in other new methodology approaches, including

data-flow techniques and evolutionary development methods, also need to be investigated for "NVN" applications.

6.5. "Non-von Neumann" Machine Assessment. Research is also needed to assess and explore the underlying architectures and application domains that encompass the class of "NVN" machines, specifically for the purpose of evaluating the impact of these architectures on the software life cycle and their potential for Air Force mission critical C3I applications. Specifically, many issues need to be addressed, such as:

1) What types of "NVN" architectures and machines currently exist? What types of "NVN" machines are being developed?

2) What types of applications and algorithms are being run on each machine? Conversely, what machines are best suited for current applications? What will be the best (or potentially the best) architectures for future applications?

3) What are the strengths and weaknesses (advantages and disadvantages) of the existing set of non-von Neumann machines?

4) What kind of success has been experienced for commercially available "NVN" machines?

5) What is the state of development for research-based "NVN" machines? Are they in a state of simulation? Do there exist prototypes? Are there plans to manufacture for commercial use?

6) What is required to support the development process of these architectures? What software is required to exploit these machines? What type of technical and financial investment is required?

These issues should provide the necessary baseline assessment of "NVN" architectures. It will be a much easier task to apply the impact of these

architectures if these issues can be understood and utilized. Until then, architecture applications cannot be made knowledgeably.

7. CONCLUSIONS.

7.1. General Conclusions. As shown in this report, parallelism can be applied in a number of ways. Summarizing we can observe that parallelism can be accomplished:

1) in an algorithm - written to express the solution to a problem,

2) in a single processor - to enhance the performance of an executable instruction,

3) in a computer system - to allow for more than one processor to solve a problem, and

4) between systems - to allow for more than one system to solve a problem.

The utilization of "NVN" architectures is definitely a very strong candidate for many current and future applications. It is apparent that what is exciting and challenging about the advent of "NVN" architectures is that they are new. Based on the understanding that is possessed by the technical community, the knowledge that has so far been obtained is only the "tip of the iceberg" [11]. Since most of the current "NVN" machines are primarily used in research labs, software for realizing the parallel nature that these machines possess is sorely needed in order to effectively utilize them in commercial and government application environments.

Once the issues portrayed in this report are successfully addressed, "NVN" architecture utilization will be more easily pursued in many areas of technology. Without this research, it will not be possible to effectively

exploit what is available, nor understand the ramifications (strategic, technical, or financial) of utilizing these machines. However, if new technology gains can be realized using "NVN" architectures, software and systems development costs may be reduced and more reliable systems may be possible.

7.2. Command and Control Utilization Observations. It is evident that "NVN" architectures are, and will continue to be, a very important avenue for providing the necessary processing power to solve many problems in a number of application areas. It is obvious that many command and control applications are suitable for exploitation by "NVN" machines. It can be observed that there exist a number of design concerns that are specifically applicable to the utilization of "NVN" computers for command and control environments:

7.2.1. Types of Processors. It appears that the types of "NVN" architectures that possibly will be employed in command and control environments will be systems built around a hybrid organization consisting of both sequential and "NVN" components. Furthermore, it appears that off-the-shelf processors, such as Intel's 80XX series, Motorola's 68XXX series, or National Semiconductor's 32XXX series may be the most promising types of processors to employ. (The math co-processors associated with each will also be employed). It is felt that these commodity level processors will be utilized because they offer the lowest cost, highest availability, and possess the most software and hardware support. The use of special purpose hardware is not desirable since they are typically expensive, hard to maintain, offer lower reliability, and offer very little initial software support. In addition to designing and developing the hardware for the system, the software must also be designed and developed. Generally off-the-shelf software packages cannot be utilized.

7.2.2. Processor Granularity. Should fine-grained architectures, which typically consist of a large number of processors, or coarse-grained

processors, which consist of relatively few (usually 2-20) computationally powerful processors, be utilized? It appears that coarse-granularity "NVN" architectures are more applicable to command and control environments. This is due to a number of observable factors. For one, course-grain machines are the most cost effective, since the processors that make-up coarse-grained systems are usually stand-alone processors. Because of this, coarse-grain machines support generic operating systems more easily. (It appears that fine-grained machines require a separate operating system processor to control such things as communication and synchronization). Maintainability of coarse-grain machines is easier, reliability is higher, graceful system degradation problems are handled easier, and interprocessor topologies are not as complicated.

7.2.3. Subset Machine Environment. A great deal of software support that is necessary for command and control environments could be done via a subset machine environment, especially if the aforementioned design criteria is employed (a coarse-grained system utilizing commodity level processors). By utilizing only a two or four processor configuration for a desired "NVN" architecture, many areas could be explored at a greatly reduced price, both in terms of cost as well as resource utilization. A "desk-top" configuration of this nature could directly support such things as software development, software testing, software tool design, simulations, and training. Obviously the low-cost distributed environment that would evolve would be very beneficial. For one, by obtaining only a subset of a desired "NVN" system, many software issues could be explored prior to the commitment to a particular system. Based upon results, a desired system may not provide the necessary processing capabilities as was thought. Hence, it is felt that the effective emulation of a full-blown system in a subset machine environment will be very beneficial. By utilizing a desk-top configuration, many command and control applications could be easily operated in a typical personal computer type of environment. For example, NCube offers a unique desk-top capability for their system.

The "guts" of their subset machine environment is a four-node board with a PC/AT bus interface that allows up to four boards. The intention by NCube is to provide a limited multiprocessing environment to more potential users.

7.3 Closing Remarks. As has been the underlying theme of this report, "NVN" processing has progressed largely because of advancements in computer architectures and hardware technology. Unfortunately, there has not been comparable investment of either research funds into the development of parallel programming languages or software support tools and techniques to utilize these technological and architectural advancements. Although the development of "NVN" computer systems are direct fallouts of hardware technology and computer architecture advancements, software development issues represent the primary roadblock to rapid implementation for these systems. In fact, hardware for "NVN" computing is probably five years ahead of software. The rather weak commercial market for the sale of "NVN" computers provides clear evidence of this. Moreover, the development and sale of commercial software for "NVN" computer systems has significantly lagged far behind the development of hardware for these systems [11]. Typically, only library subroutines that assist a programmer to perform parallel operations are provided by today's manufacturer's, but still the programmer usually must modify their particular computer programs to take advantage of them [28]. It has taken more than three decades to build the foundation for sequential processing, yet it is very unclear as to whether any of this previously developed software technology baseline can be applied or extended to "NVN" machines. It appears that sequential processing can be used as a starting point, but major technological breakthroughs in software techniques are sorely needed.

Hopefully, this report has provided a better understanding of "NVN" architectures and the underlying software issues that surround their effective implementation. If nothing else, it is hoped that the reader has obtained at least some kind of knowledgeable insight into many of the

areas of "NVN" computing. Possibly, this report has fueled an interest to obtain further specific information on "NVN" processing. If so, the reader is advised to utilize the reference list at the conclusion of this report.

7.4. Acknowledgements. The author would like to thank Denis Maynard and Joe Cavano for their constructive criticisms, continued support, and careful review of various portions of this report. Also, thanks to Chris Flynn, Pat O'Neill, Mark Foresti, and Bob Vaeth for their helpful services in reviewing the final draft of this report.

8. REFERENCES.

[1] Allen, J. R. and Kennedy, K. "A Parallel Programming Environment," IEEE Software, July 1985, pp. 21-29.

[2] Almasi, G. S. "Overview of Parallel Processing," Parallel Computing, Volume 2, Number 3, November 1985, pp. 191-203.

[3] Anderson, T. and Lee, P. A. Fault Tolerance: Principles and Practice, Prentice-Hall International, London, 1981.

[4] Andrews, G. R. and Schneider, F. B. "Concepts and Notations for Concurrent Programming," Computing Surveys, Volume 15, Number 1, March 1983, pp. 3-42.

[5] Artym, R. and Mason, J. S. "XPXM/C: A Taxonomy of Processor Coupling Techniques," IEEE Proceedings, Volume 135, Part E, Number 3, May 1988, pp. 173-179.

[6] Baer, J.-L. Computer Systems Architecture, Computer Science Press, Inc., Rockville, Maryland, 1980.

[7] Bell, C. G., Miranker, G. S., and Rubenstein, J. J. "Supercomputing for One," IEEE Spectrum, April 1988, pp. 46-50.

[8] Browne, J. C. "Framework for Formulation and Analysis of Parallel Computation Structures," Parallel Computing, Volume 3, 1986, pp. 1-9.

[9] Burks, A. W., Goldstine, H. H., and von Neumann, J. "Preliminary Discussion of the Logical Design of an Electronic Computing Device," Collected Works of John von Neumann, Volume 5, The Macmillan Company, New York, 1963, pp. 34-79, taken from report to U.S. Army Ordinance Department, 1946.

[10] Butler, J. M. and Oruc, A. Y. "A Facility for Simulating Multiprocessors," IEEE Micro, October 1986, pp. 32-44.

[11] Davis, D. B. "Parallel Computers Diverge," High Technology, February 1987, pp. 16-22.

[12] DOD-STD-2167A, Defense System Software Development, United States Department of Defense, 29 February 1988.

[13] Feng, T.-Y. and Wu, C.-L. "Interconnection Networks in Multiple Processor Systems," RADC Technical Report 79-304, December 1978.

[14] Fox, G. C., and Messina P. C. "Advanced Computer Architectures," Scientific American, October 1987, pp. 67-74.

[15] Gajski, D. D. and Peir, J.-K. "Essential Issues in Multiprocessor Systems," Computer, June 1985, pp. 9-27.

[16] Gelernter, D. "Programming for Advanced Computing," Scientific American, October 1987, pp. 91-98.

[17] Gustafson, J. L. "Reevaluating Amdahl's Law," Communications of the ACM, Volume 31, Number 5, May 1988, pp. 532-533.

[18] Hall, W. and Stavely-Taylor, B. "An IKBS to Support the Learning of Ada as a Second Programming Language," Journal of Pascal, Ada, and Modula-2, Volume 7, Number 1, 1988, pp. 32-35.

[19] Hockney, R. W. "MIMD Computing in the USA - 1984," Parallel Computing, Volume 2, 1985, pp. 119-136.

[20] Howe, C. D. and Moxon, B. "How to Program Parallel Processors," IEEE Spectrum, September 1987, pp. 36-41.

[21] Hwang, K. "Advanced Parallel Processing with Supercomputer Architectures," Proceedings of the IEEE, Volume 75, Number 10, October 1987, pp. 1348-1379.

[22] Hwang, K. and Briggs, F. A. Computer Architecture and Parallel Processing, Mc-Graw Hill, Inc, New York, 1984.

[23] Jamieson, L. H., Siegel H. J., Delp E. J., and Whinston, A. "The Mapping of Parallel Algorithms to Reconfigurable Parallel Architectures," Proceedings of the ARO Workshop on Future Directions in Computer Architecture and Software, Charleston, South Carolina, May 1986, pp. 147-154.

[24] Jamieson, L. H. "Features of Parallel Algorithms," Proceedings on the 2nd International Conference on Supercomputers, San Francisco, California, May 1987.

[25] Jones, A. K. and Schwarz, P. "Experience Using Microprocessor Systems - A Status Report," Computing Surveys, Volume 12, Number 2, June 1980, pp. 121-165.

[26] Kuck, D. J. The Structures of Computers and Computations, John Wiley and Sons, Inc, New York, 1978.

[27] Kutti, S. "Taxonomy of Parallel Processing and Definitions," Parallel Computing, Volume 2, Number 3, November 1985, pp.353-359.

[28] Lehman, M. "A Survey of Problems and Preliminary Results Concerning Parallel Processing and Parallel Processors," Proceedings of the IEEE, Volume 54, Number 12, December 1966, pp. 1889-1901.

[29] Maples, C "Analyzing Software Performance in a Multiprocessor Environment," IEEE Software, July 1985, pp. 50-63.

[30] Marzjarani, M. "A Comparison of the Computer Languages Pascal, C, Lisp, and Ada," Journal of Pascal, Ada, and Modula-2, Volume 7, Number 1, 1988, pp. 5-10.

[31] Miller, R. and Stout, Q. F. "Geometric Algorithms for Digitized Pictures on a Mesh-Connected Computer," IEEE Transactions on Pattern Analysis and Machine Intelligence, Volume PAMI-7, Number 2, March 1985, pp. 216-228.

[32] Miller, S. A. "A Survey of Parallel Computing," RADC Technical Report 88-129, July 1988.

[33] Myers, W. "Getting the Cycles out of a Supercomputer," IEEE Computer, March 1986, pp. 89-100.

[34] Olson, R. A. "Parallel Processing in a Message-Based Operating System," IEEE Spectrum, July 1985, pp. 39-49.

[35] Peled, A. "The Next Computer Revolution," Scientific American, Volume 257, Number 4, October 1987, pp. 57-64.

[36] Quinn, M. J. Designing Efficient Algorithms for Parallel Computers, McGraw-Hill, Inc, New York, 1987.

[37] Reiss, S. P. "Programming Environments Today," Annual Review of Computer Science, Volume 1, 1986, pp. 181-195.

[38] Schneck, P. B., Austin, D., Squires, S. L., Lehmann, J., Mizell, D., and Wallgren, K. "Parallel Processor Programs in the Federal Government," IEEE Computer, July 1985, pp. 43-56.

[39] Schwederski, T., and Siegel H. J. "Adaptable Software for Supercomputers," Computer, Vol.19, No. 2, pp.40-48, February 1986.

[40] Schwederski, T., Meyer D. G., and Siegel H. J. "Parallel Processing," Computer Architecture: Conceptal Systems, Elsevier Science Publishing Co, Inc, New York, 1987.

[41] Seitz, C. L. "The Cosmic Cube," Communications of the ACM, January 1985, Volume 28, Number 1, pp. 22-33.

[42] Siegel, H. J., and Hsu W. T. "Interconnection Networks," Computer Architecture: Concepts and Systems, edited by V. M. Milutinovic, Elsevier Science Publishing Co, Inc, New York, 1987.

[43] Shinners, S. M. "Which Computer . . . Analog, Digital, or Hybrid?," Machine Design, Jan 21, 1971, pp. 104-111.

[44] Snyder, L. "Introduction to the Configurable, Highly Parallel Computer," IEEE Computer, January 1982, pp. 47-56.

[45] Yalamanchili, S. and Aggarwal, J. K. "Reconfiguration Strategies for Parallel Architectures," IEEE Computer, December 1985, pp. 44-61.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.